UNIVERSITA' DEGLI STUDI DI MODENA E REGGIO EMILIA
FACOLTA' DI INGEGNERIA – SEDE DI MODENA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

SVILUPPO DI APPLICAZIONI WEB-DB CON ARCHITETTURE A DUE E TRE LIVELLI: VALUTAZIONE COMPARATIVA

ELABORATO DI LAUREA DI FILIPPO BATTILANI

RELATORE:
CHIAR.MO PROF. SONIA BERGAMASCHI

ANNO ACCADEMICO 2001-2002

Ringraziamenti

Un particolare ringraziamento va alla Prof. Sonia Bergamaschi, relatore di questa tesi, per i preziosi suggerimenti forniti e la collaborazione per la stesura del lavoro.

Desidero inoltre ringraziare tutte le persone care per la fiducia, il supporto e l'affetto profusi durante tutti gli anni della mia vita.

INDICE

Introduzione

Il presente elaborato consiste in una trattazione dettagliata inerente le architetture di sistema sulle quali possono basarsi le applicazioni Web-DB; in particolare si fa riferimento alle architetture a due e tre livelli e alla piattaforma Java 2 Enterprise Edition (J2EE).

L'analisi si integra alla tesi di diploma "Un sito internet per la gestione dei progetti" da me realizzata durante il periodo di stage presso la ditta Intertech Italia s.n.c. di Modena. Lo strumento a cui si riferisce la tesi sopracitata, denominato Managing Information System Projects (MISP), si traduce in un'applicazione Web-DB two-tier che si occupa della gestione e dello sviluppo dei progetti.

L'obiettivo che si intende pertanto raggiungere è approfondire l'aspetto tecnico della tesi ed evidenziare come sia possibile realizzare un'applicazione Web-DB del tipo di MISP con un'architettura a tre livelli, invece che con una a due, indicandone i vantaggi e la maggiore complessità.

L'elaborato è composto da due capitoli. Il primo richiama l'origine e le caratteristiche principali di MISP descrivendone il funzionamento.

Il secondo capitolo, invece, illustra in modo dettagliato le funzionalità e le potenzialità fornite dalla piattaforma J2EE analizzandone il modello di applicazione distribuita multilivello.

Seguono le conclusioni e la bibliografia.

Capitolo 1

Managing Information System Projects (MISP)

1.1 Nascita dell'applicazione

MISP è il risultato del lavoro realizzato durante il periodo di stage presso la ditta Intertech Italia s.n.c. di Modena, e consiste in un prototipo di un'applicazione per la gestione e lo sviluppo di progetti Internet.
Intertech Italia, infatti, opera da tre anni nel campo dello sviluppo di siti e applicazioni Intranet/Internet rivolti ad un'utenza aziendale di livello medio-alto, appartenente sia all'area pubblica sia a quella privata. La creazione di siti e applicazioni è un'attività complessa, perché "portare un'azienda in rete" significa creare un biglietto da visita osservabile in tutto il mondo.
In seguito a numerosi fattori, tra i quali la richiesta di progetti sempre più impegnativi, Intertech Italia è stata investita da un'importante crescita accompagnata da incrementi del personale. E' per questo che è emersa la necessità di coordinare maggiormente il lavoro di gruppo, sia all'interno dei gruppi stessi, ovvero tra le persone che lavorano sullo stesso progetto, sia tra gruppi di lavoro diversi.
L'obiettivo che si è voluto pertanto conseguire consiste nella realizzazione di uno strumento in grado di standardizzare, organizzare e condividere secondo logiche di lavoro in team le informazioni riguardanti i progetti di Intertech Italia, rendendo semplice e veloce il reperimento e l'accesso a tali informazioni secondo precisi criteri di visibilità (non tutti possono vedere) e privilegi (non tutti possono fare).
Per essere vincente uno strumento di questo tipo deve inserirsi armoniosamente all'interno del sistema informativo aziendale, ovvero deve integrarsi con gli strumenti di lavoro già esistenti (ad es.: CVS = Concurrent Version System), consentendo in tale modo agli utenti di continuare a lavorare in modo naturale, ma al tempo stesso in maniera più efficiente. L'interfaccia grafica e la struttura intuitiva hanno reso infatti MISP immediatamente utilizzabile da parte di qualsiasi utente, che può in questo modo operare in maniera autonoma, senza modificare il modo di lavorare abituale.
L'interesse per applicazioni di questo tipo denota, specie nelle grandi aziende, una maggiore attenzione verso i propri dipendenti, che hanno esigenze e necessità precise da soddisfare. Il risultato è il consolidamento di una metodologia di lavoro, tesa ad identificare senza ambiguità soggetti e compiti e a raggiungere un maggior grado di soddisfazione sia da parte del cliente esterno che ha commissionato il progetto Internet sia da parte del dipendente, che inizia ad assumere in quest'ottica il nuovo ruolo di "cliente interno".

1.2 Architettura di sistema

1.2.1 Il funzionamento di MISP

L'applicazione MISP consiste in un sito internet "dinamico" le cui pagine vengono generate utilizzando le informazioni memorizzate all'interno di un database relazionale e manipolandole nel modo opportuno. Accanto alle tradizionali possibilità a disposizione delle aziende che sviluppano siti Internet (ASP, PERL, CGI e PHP) sta prendendo sempre più piede l'utilizzo di applicazioni Java, il linguaggio di programmazione ad oggetti sviluppato dalla Sun Microsystem. Oltre all'applet, eseguita direttamente dal browser dell'utente (**applicazioni web browser-side**), uno degli standard emergenti per lo sviluppo di applicazioni client/server è rappresentato dalle applicazioni Java/JSP eseguite dal server web (**applicazioni web server-side**).

Il funzionamento di un sistema che prevede l'utilizzo di applicazioni Java/JSP è semplice e risulta molto simile a quello di un normale motore di ricerca. I parametri di ricerca, introdotti dall'utente all'interno del browser Internet (Figura 1), vengono inviati al web server (Figura 2) che, riconosciuta la richiesta, la inoltra direttamente all'applicazione Java/JSP (Figura 3).

Figura 1

Figura 2

Figura 3

L'applicazione Java/JSP genera dinamicamente una o più interrogazioni, chiamate tecnicamente "query" (Figura 4), che vengono inviate al database server a cui si appoggia l'applicazione. Il database server elabora i dati (Figura 5), restituendo all'applicazione Java/JSP le informazioni richieste (Figura 6).

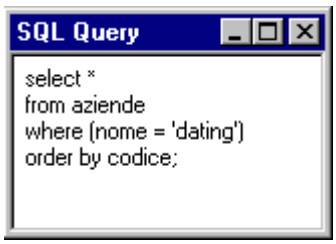


Figura 4

Figura 5

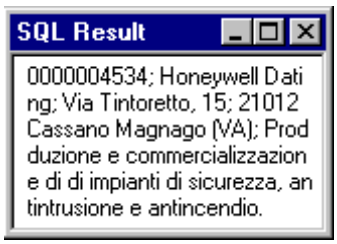


Figura 6

L'applicazione Java/JSP (Figura 7) è in grado di creare dinamicamente una pagina Internet contenente i risultati ottenuti dal database e di inviarla al web server (Figura 8) che a sua volta la fornisce all'utente che ne ha fatto richiesta. Il risultato finale viene quindi mostrato all'utente sotto forma di pagina Internet all'interno del proprio browser (Figura 9).

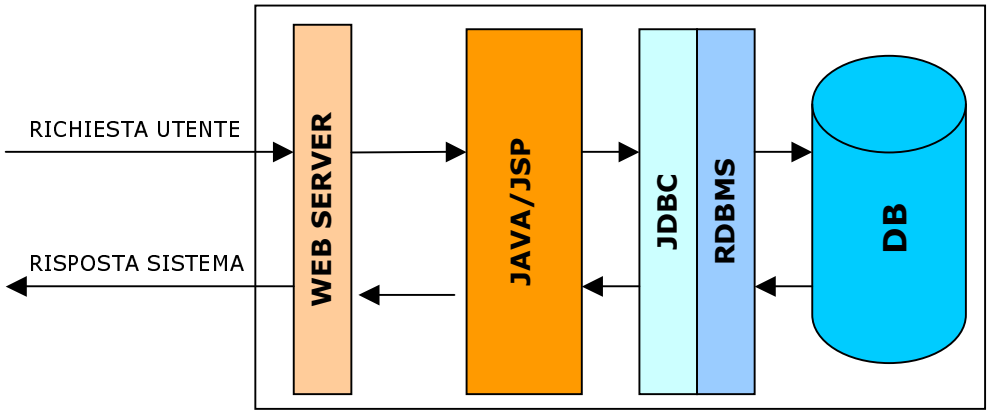
Figura 7

Figura 8

Figura 9

1.2.2 Layout di sistema

Tutto il software viene installato su **un'unica macchina** che in questo modo assume il ruolo sia di web server che di database server.



WEB SERVER/DATABASE SERVER

Figura 10

1.2.3 Vantaggi e limiti dell'architettura a due livelli di MISP

Lo sviluppo di applicazioni funzionanti nella modalità sopra descritta non richiede particolari conoscenze e competenze da parte dello sviluppatore, e consente di progettare le stesse in tempi relativamente contenuti ottenendo risultati decisamente soddisfacenti sotto diversi punti di vista.

Per realizzare un'applicazione di alta complessità, però, un'architettura come quella di MISP non garantisce una sufficiente strutturazione e può risultare limitante. E' per questo che nel capitolo seguente verranno illustrate in modo dettagliato la piattaforma J2EE e le relative specifiche sulle quali si basano le applicazioni Web-DB realizzate in linguaggio Java. Si potranno notare i numerosi servizi che la piattaforma mette a disposizione degli sviluppatori di applicazioni di questo tipo e la possibilità di realizzare architetture a **tre livelli**.

Capitolo 2

La piattaforma J2EE

L'obiettivo della piattaforma Java 2 Enterprise Edition (J2EE), distribuita della Sun Microsystem, consiste nel definire funzionalità standard che supportino gli sviluppatori nella realizzazione di applicazioni distribuite, ovvero degli involucri attraverso i quali vengono trasmesse le informazioni verso il mondo di internet. Durante la trattazione si cercheranno di mettere in evidenza le numerose potenzialità che l'architettura a **tre livelli** prevista nelle specifiche J2EE presenta rispetto ad una più semplice architettura a **due livelli**.
All'interno del capitolo sono stati inseriti anche alcuni semplici esempi con la finalità di rendere più facile al lettore la comprensione dei concetti chiave.

2.1 Vantaggi della piattaforma J2EE

La piattaforma J2EE offre una serie di benefici agli sviluppatori di applicazioni distribuite.

2.1.1 Architettura e sviluppo semplificati

La J2EE supporta un modello di sviluppo *component-based* semplificato. Essendo basata sul linguaggio di programmazione Java e sulla Java 2 Standard Edition (J2SE platform), questo modello offre grandi vantaggi di portabilità (sistema *Write Once*, *Run Anyway*) e garantisce il supporto su tutti i server conformi a questo standard.
Il modello di sviluppo component-based J2EE arricchisce la produttività delle applicazioni in diversi modi:

1. assicura flessibilità alle funzionalità desiderate, consentendo diverse possibilità di configurazione dell'architettura della applicazione, a seconda di vari fattori, come ad esempio il tipo di client, la sicurezza richiesta per gli accessi alle sorgenti di dati, e altri che verranno approfonditi in seguito. Il modello component-based inoltre semplifica la manutenzione dell'applicazione, in quanto i singoli componenti possono essere aggiornati o sostituiti indipendentemente.
2. assicura ai componenti la disponibilità di una serie di servizi nell'ambiente di run-time, e la possibilità di essere dinamicamente connessi ad altri componenti, semplicemente fornendo alcune interfacce. Ad esempio attraverso i *deployment descriptors* (un file di testo in formato XML che specifica il comportamento di un componente attraverso tag XML) è possibile comunicare specifici parametri all'ambiente di run-time personalizzando l'applicazione senza dover modificare il codice dei componenti.
3. supporta la suddivisione dei compiti, in quanto ciascun set di componenti può essere associato ad un certo ambito di sviluppo, consentendo a ciascuna figura professionale di concentrarsi specificatamente sulle proprie competenze e abilità. Questa suddivisione dei compiti, oltre a favorire la specializzazione e quindi migliorare la qualità dei singoli componenti, consente un criterio di sviluppo dell'applicazione in parallelo, aumentandone la velocità di produzione e manutenzione.

2.1.2 Scalabilità

La J2EE fornisce supporti per adattare in modo semplice e funzionale una applicazione ad eventuali aumenti o riduzioni delle dimensioni del progetto senza comprometterne l'efficienza delle prestazioni. Ad esempio il supporto per il *connection pooling* (il pool di connessioni è un meccanismo molto utile per diminuire il gravoso tempo di accesso ai database) assicura rapido accesso ai dati anche ad un numero crescente di clients.
Inoltre l'architettura distribuita che la J2EE supporta consente di ottenere un vantaggioso sistema di bilanciamento del carico di lavoro, permettendo la configurazione dei vari livelli dell'architettura per la gestione du server diversi.

2.1.3 Integrazione su sorgenti di informazione già esistenti

La J2EE platform, assieme alla J2SE platform, include una serie di *API* (*Application Program Interface*) standard per accedere alle varie sorgenti di informazione esistenti nell'impresa.
Si riporta l'elenco di queste API.

- *JDBC* (*Java Database Connectivity*) è l'API per la connessione ai database relazionali. Essa fornisce un'interfaccia a livello di applicazione usata nel codice dei componenti per accedere al database in modo indipendente dal particolare DBMS utilizzato; sarà poi necessario utilizzare un driver specifico che si ponga come interfaccia tra l'API JDBC e lo specifico DBMS utilizzato. Grazie a questo meccanismo a due livelli è possibile mantenere il codice dei componenti dell'applicazione indipendente dal DBMS che può essere sostituito con un altro semplicemente cambiando il driver, e senza necessità di riscrittura del codice.
- *JTA* (*Java Transaction API*) è l'API per la gestione e il coordinamento delle transazioni attraverso sorgenti di informazioni transazionali eterogenee.
- *JNDI* (*Java Naming and Directory Interface*) è l'API per accedere ad informazioni attraverso un sistema di nomi e percorsi, utilizzando così lo stesso meccanismo utilizzato per riferire i files nel File System.
- *JMS* (*Java Message Service*) è l'API per inviare e ricevere messaggi attraverso sistemi di enterprise-messaging come l'IBM MQ Series e TIBCO Rendez-vous che richiedono l'attivazione del servizio attraverso un JMS provider.
- *JavaMail* è l'API utilizzata dai componenti per inviare e ricevere email.
- *Java IDL* è l'API per richiamare oggetti CORBA remoti, attraverso il protocollo IIOP. Questi oggetti sono tipicamente esterni alla J2EE e possono essere scritti in qualsiasi linguaggio.

Oltre a questi servizi che già sono presenti nell'attuale versione della J2EE platform, ve ne sono altri in via di sviluppo che saranno aggiunti nelle successive versioni attraverso l'architettura dei *Connectors*.

2.1.4 Scelta di servizi, tools e componenti

Il marchio J2EE è punto centrale per creare un mercato di servizi, tools e componenti tutti conformi allo stesso standard.
Le organizzazioni che sviluppano applicazioni J2EE possono contare su una varietà sempre crescente di fornitori di piattaforme J2EE con diverse possibilità di scelta sia a livello hardware, sia a livello di sistemi operativi e configurazioni di server, a seconda degli obiettivi e strategie adottate.
I vari componenti J2EE, con particolare riferimento agli EJB e alle JSP di cui si tratterà dettagliatamente in seguito, sono stati ideati per essere facilmente manipolati da tool grafici che consentono di automatizzare tanti dei tradizionali compiti richiesti, come ad esempio il debugging. Gli sviluppatori J2EE hanno così la possibilità di scegliere il tool conforme allo standard J2EE che meglio si addice alle loro esigenze.
Grazie al modello component-based si ha poi la possibilità di sviluppare componenti conformi allo standard che possono essere così riutilizzati da una qualsiasi applicazione J2EE.

2.1.5 Modello di sicurezza semplificato e unificato

Gli sviluppatori possono specificare i requisiti di sicurezza di un componente. Attraverso un sistema dichiarativo, sia gli EJB-component sia i Web-component possono specificare attraverso i deployment descriptor i criteri per associare i vari livelli di accesso a diversi ruoli e quindi alle diverse categorie di utenti. Per gli EJB esiste addirittura la possibilità di definire ruoli diversi per ciascun metodo.
Le specifiche J2EE incoraggiano l'utilizzo di questo sistema dichiarativo per consentire al codice dei componenti di rimanere totalmente indipendente dai vincoli legati alla sicurezza. Ciò non sempre è possibile, poiché in taluni casi è necessario creare livelli e vincoli di sicurezza che non possono essere espressi con un semplice mapping tra ruoli e utenti. Perciò viene comunque mantenuta la possibilità di inserire security-checks anche all'interno del codice attraverso opportune API.

2.2 Architettura multilivello della J2EE

Si analizza ora nel dettaglio il modello di applicazione distribuita multilivello della J2EE platform.
Si è già accennato alla caratteristica component-based del modello J2EE con i vantaggi che ne derivano. La logica di ogni applicazione J2EE può essere infatti suddivisa in più componenti in accordo con la funzione da essi svolta, ciascuno dei quali può essere installato in una macchina differente a seconda del livello logico di appartenenza all'interno dell'architettura J2EE.
L'architettura multilivello J2EE individua tre livelli fondamentali: il *client-tier*, il *middle-tier* e l'*EIS-tier*.

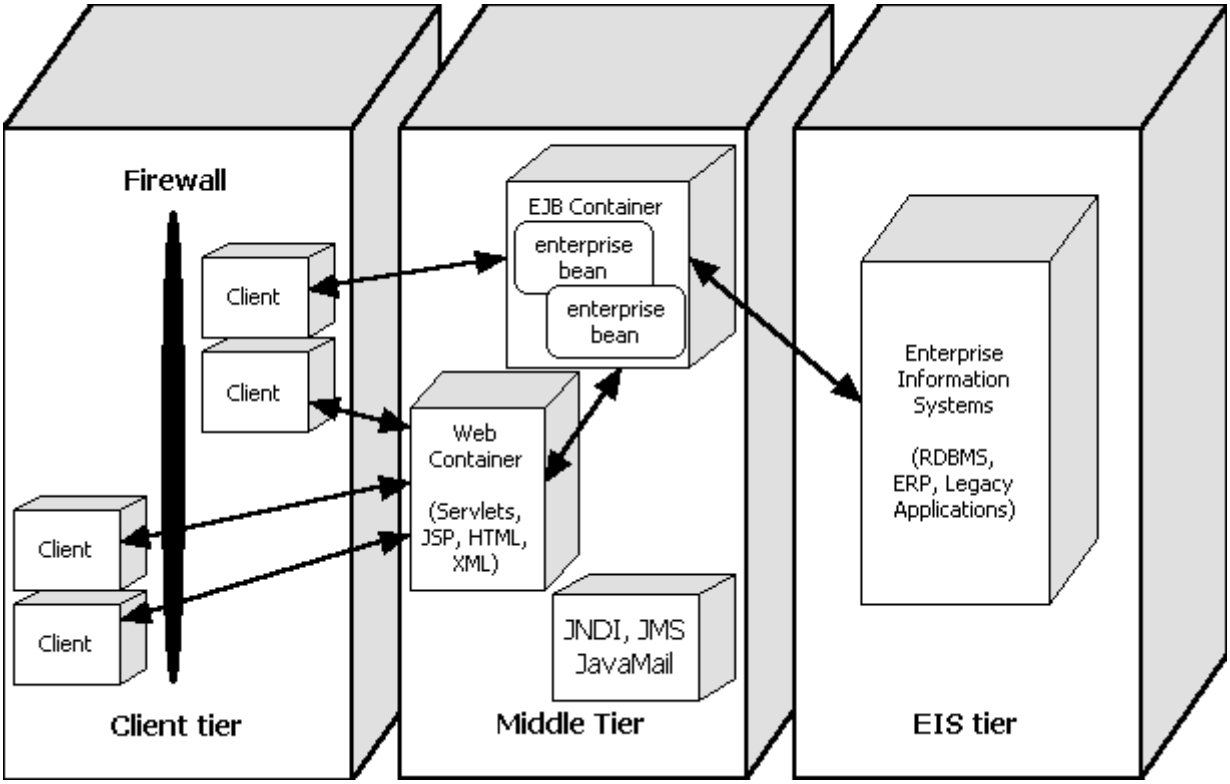


Figura 11-Architettura J2EE

La figura mostra i vari componenti e servizi che costituiscono un tipico ambiente J2EE. La presenza di un livello intermedio tra il client e le sorgenti di informazione fornisce alla J2EE platform i tipici vantaggi derivanti da un'architettura multilivello.

Le tipiche architetture client-server a due livelli, nelle quali il server è in pratica costituito dalle sorgenti di informazione, hanno dato prova di essere fortemente limitanti a causa della necessità di installare e mantenere un front-end completo di interfaccia grafica e di logica applicativa direttamente su ciascun client. Ciò naturalmente si traduce in un forte carico di lavoro sulla macchina client, creando problemi derivati dalla limitatezza delle risorse di tali macchine, tipicamente dei semplici PC, e nel non meno grave problema della manutenzione dell'applicazione: ogni più piccolo cambiamento nel codice dell'applicazione deve essere ripetuto su tutti i client.

Con un'architettura a tre livelli questi problemi vengono superati grazie all'introduzione di un middle tier che si pone tra il client e la sorgente di informazioni. In questo scenario il server può ospitare sia il middle tier che l'EIS tier, oppure questi ultimi possono essere ospitati su server differenti, dando origine al middle tier server e all'EIS server.

Il middle tier server si occupa di gestire le richieste dei vari client, di reperire le opportune informazioni dall'EIS server, rielaborarle secondo la logica applicativa, e fornirle al client, che si trova così notevolmente alleggerito.

La J2EE fornisce uno standard ben preciso per implementare il middle tier server, e fornisce indicazioni e raccomandazioni per implementare gli altri due livelli.

Si rimanda ai prossimi paragrafi la descrizione dettagliata dei tre livelli della J2EE, mentre in questa sede si vuole introdurre un altro elemento chiave dell'architettura: i *containers*.

I containers sono degli ambienti run-time standardizzati in grado di fornire specifici servizi ai vari componenti. Ciascun componente può contare sul supporto del container in cui è inserito per ottenere gli stessi servizi standard in tutte le piattaforme J2EE qualunque sia lo specifico provider. I vantaggi che ne derivano sono i seguenti:

- gli sviluppatori possono disinteressarsi di come i container gestiscono tali servizi, potendo così concentrarsi sulla funzionalità del codice.
- attraverso i deployment descriptor, che fungono da interfaccia tra i componenti e il container si possono specificare i parametri e i comportamenti dei componenti stessi al momento del deployment dell'applicazione senza dover interferire sul codice.

2.3 EIS (Enterprise Information System) tier

E' il livello più basso dell'architettura, cioè la parte di dati memorizzati in modo permanente con varie tipologie di risorse, tipicamente RDBMS. Questi ultimi possono supportare accessi controllati attraverso le transazioni. Essi possono partecipare in transazioni assieme ad altre risorse transazionali in un sistema di database distribuito attraverso il protocollo *two-pase commit*, gestito da un transaction manger supportato dal J2EE server.
Questa integrazione di sorgenti di informazione funziona bene quando queste sono omogenee. Purtroppo non sempre è così quando si sviluppa un'applicazione distribuita che si deve appoggiare a varie sorgenti.
E' attualmente in via di sviluppo da parte dei progettisti della Sun un nuovo tipo di API, la *API J2EE Connector*, allo scopo di definire uno standard per connettere la J2EE platfom con sorgenti di informazione eterogenee.

2.4 Middle tier

I maggiori benefici del modello di applicazioni J2EE si riscontrano nel middle tier. Esso è ulteriormente scomposto in due sottolivelli: un *Web-tier* e un *EJB-tier* che possono trovarsi anche su differenti hosts, oppure sullo stesso host ma in differenti Java Virtual Machines, o infine anche sulla stessa JVM.
Le funzioni di business logic della applicazione vengono implementate attraverso componenti *Enterprise Java Beans (EJB)* all'interno dell'EJB-tier.
Al Web-tier spetta invece il compito di "presentare" al client (in questo caso un Web-client) i risultati dell'elaborazione della business logic, attraverso la generazione di pagine web dinamiche.

2.4.1 EJB tier

La tecnologia degli *Enterprise Java Beans (EJB)* [SPECEJB] offre un modello component-based distribuito che consente agli sviluppatori di concentrare la loro abilità sui business problem dell'applicazione, lasciando al *EJB container* il compito di gestire tutta una serie di servizi di sistema a basso livello. Questa separazione di ruoli, punto di forza dell'architettura J2EE che si ritrova anche nel Web tier, permette un più rapido sviluppo di applicazioni altamente scalabili, flessibili e sicure.

Gli EJB sono componenti che costituiscono un collegamento fondamentale tra i componenti del Web tier ai quali viene affidato esclusivamente il compito di gestire la presentation logic, e le risorse di informazione mantenute nel EIS tier.

La Business logic e i business-objects

E' difficile dare una definizione rigorosa di business logic di una applicazione. In senso lato può essere definita come un insieme di direttive per gestire ogni specifica funzione che l'applicazione deve svolgere.
Volendo adottare un approccio object-oriented, una funzione può essere scomposta in un insieme di componenti, o elementi chiamati *business-objects*. Come gli altri elementi, anche questi sono dotati di una specifica struttura dati e di un determinato comportamento. Ad esempio un impiegato può essere modellato con un oggetto impiegato che ha una struttura dati, rappresentata da un insieme di attributi quali il nome, l'indirizzo, il codice fiscale e così via; inoltre ha dei metodi ad es. per assegnarlo ad un nuovo dipartimento, aumentargli lo stipendio, ecc.
In modo più rigoroso si può allora definire la business-logic come l'insieme delle regole che servono ad identificare struttura e comportamento dei business-objects, e i criteri di interazione con gli altri oggetti dell'applicazione.

Requisiti comuni dei business-objects

Si riportano qui di seguito alcuni requisiti comuni ai business object.

1. *Mantenere lo stato*. I business-objects solitamente necessitano di mantenere, in modo conversazionale o permanente, lo stato rappresentato dalle variabili di istanza tra le invocazioni dei vari metodi.
2. *Operare su dati condivisi.* La condivisione di certe risorse deve essere gestita attraverso il controllo della concorrenza e un appropriato livello di isolation dei dati condivisi, in modo da assicurarne la consistenza.
3. *Partecipare alle transazioni.* L'atomicità di una transazione deve essere garantita anche se questa si distribuisce su diverse risorse remote.
4. *Servire un gran numero di client.* Un business-object deve essere disponibile in istanze multiple a diversi client contemporaneamente. Il meccanismo di gestione di queste istanze deve essere in grado di fornire a ciascun client un business-object disponibile a servire la sua richiesta. Senza un tale meccanismo il sistema potrebbe eventualmente esaurire le risorse e non essere e non essere più in grado di servire ulteriori richieste.
5. *Fornire accesso remoto ai dati.* I client devono poter accedere alle risorse dei business-objects anche da remoto attraverso la rete.
6. *Controllare gli accessi.* I servizi offerti dai business-objects spesso richiedono l'autenticazione del client che ne fa richiesta, per consentire l'accesso a risorse protette ai soli client autorizzati.
7. *Garantire la ricusabilità.* E' questo un requisito comune a tutti i tipi di oggetti in un approccio object-oriented.

Gli Enterprise Java Beans e l'EJB Container

Nell'architettura J2EE i business-objects vengono modellati con componenti Enterprise Java Beans. Come già accennato essi possono contare sul supporto fornito dall'EJB container che ne gestisce il ciclo di vita e fornisce loro una varietà di servizi. Quando un client invoca un'operazione su un EJB questa viene prima intercettata dal container che può in tal modo gestire i vari servizi che devono eventualmente propagarsi tra diverse sottochiamate ad altri componenti dell'EJB tier.

Grazie all'intercessione del container è anche possibile definire determinati comportamenti del componente al momento del deployment a seconda delle esigenze senza dover effettuare alcun cambiamento nel codice e ricompilazione, ma semplicemente configurandoli attraverso i deployment descriptor.

Il Bean Provider, cioè la figura professionale che si occupa di sviluppare questi componenti, deve fornire anche una "client wiew" del componente stesso attraverso la definizione di due interfacce: la *home interface* e la *remote interface*. Questo assicura che il client abbia una visione semplificata del componente e che possa disinteressarsi dei dettagli implementativi. Sarà compito del container implementare queste interfacce, crearne le istanze e gestirne il ciclo ci vita.

La *home interface* fornisce i metodi per creare e rimuovere gli EJB. Essa deve estendere l'interfaccia `javax.ejb.EJBHome`. Attraverso la home interface il client può inoltre ottenere informazioni sui meta-dati dell'EJB. Per particolari tipi di EJB, di cui si parlerà di seguito, la home interface consente anche di ottenere un handle, cioè un riferimento, alla interfaccia stessa che può essere e salvato in modo permanente, e infine consente di recuperare istanze particolari del bean salvate in precedenza.

La *remote interface* definisce la vera e propria "client wiew" dell'EJB, cioè il set di business methods disponibili ai clients. Essa deve estendere l'interfaccia `javax.EJB.EJBObject`.

Per completare lo sviluppo di un EJB occorre fornire anche l'implementazione attuale dei business methods del bean. Ciò viene fatto fornendo la *Enterprise Bean class* che deve contenere il codice di tutti i metodi dichiarati nella remote interface e inoltre deve contenere un metodo `ejbCreate` per ciascun metodo create della home interface. Quando il client invoca un metodo della remote interface, il container lo intercetta e richiama il corrispondente metodo della classe del bean.

L'Enterprise Bean class deve estendere l'interfaccia `javax.ejb.EntityBean` oppure `javax.ejb.SessionBean`, a seconda del tipo di bean che implementano.

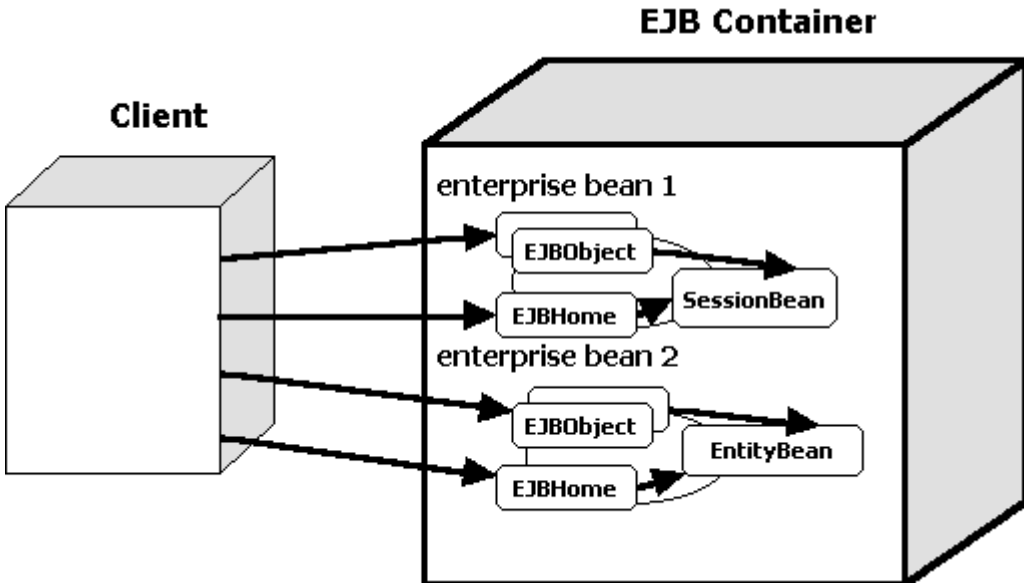


Figura 12-Implementazione della client wiew di un EJB

Session beans

Come suggerisce il nome, un *session bean* è simile ad una sessione di interazione con un particolare client. Infatti un session bean rappresenta un unico client all'interno del J2EE server, e non è condiviso da nessun altro client. Esso non è persistente, cioè quando termina la sessione del client ogni suo riferimento è perso e le risorse associate ad esso sono rilasciate.

Esistono due categorie di session bean: lo *stateful session bean* e lo *stateless session bean*.

Lo *stateful session bean* contiene lo stato conversazionale del client, che significa che lo stato viene mantenuto per tutta la durata della sessione. Per stato conversazionale si intende non solo i valori delle variabili di istanza, ma anche gli oggetti raggiungibili attraverso tali variabili.

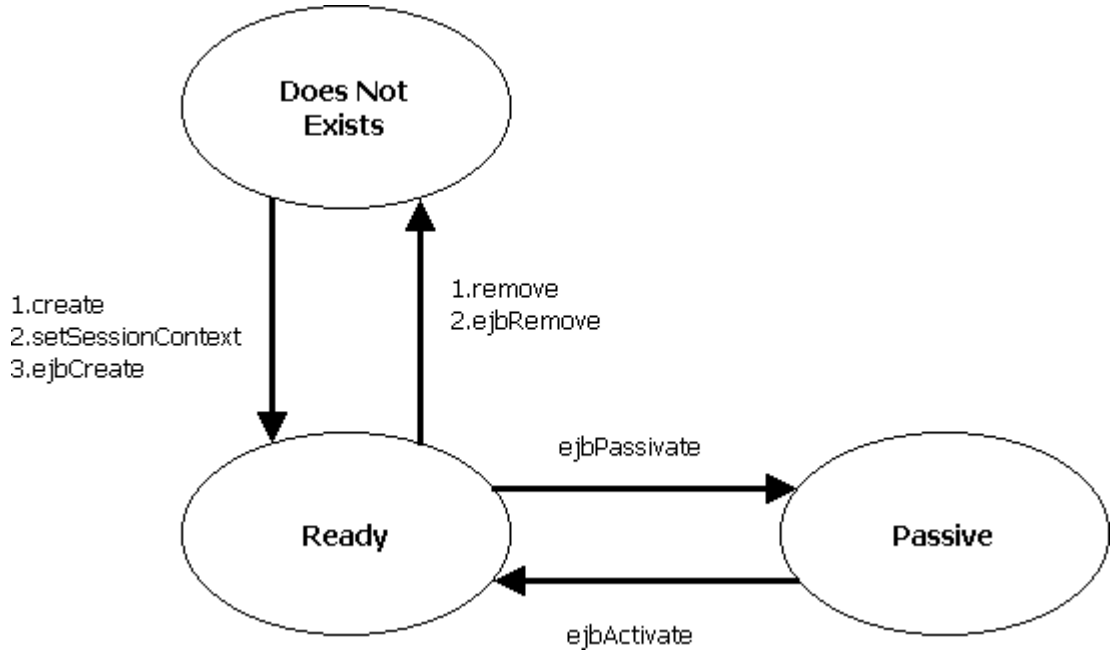


Figura 13-Ciclo di vita di uno stateful session bean

La figura 13 mostra il ciclo di vita dello stateful session bean. Il client inizia il ciclo di vita richiamando il metodo create della home interface. Il container intercetta questa invocazione e provvede ad effettuare altre due operazioni: invoca il method `setSessionContext` che setta i parametri della sessione nel bean (ad esempio gli eventuali parametri di autenticazione) e richiama il metodo `ejbCreate` della classe del bean corrispondente a quello invocato dal client. A questo punto il bean si trova nello stato di "pronto" a ricevere le invocazioni dei suoi metodi. Quando il client invoca un metodo della remote interface, questo viene intercettato dal container che invoca il corrispondente metodo della classe del bean e il bean rimane in stato "pronto".

Per non appesantire troppo il sistema il container può decidere di liberare la memoria degli stateful session bean che si trovano in stato di pronto ma non vengono riferiti da un certo periodo di tempo, e salvarli in memoria secondaria, passandoli così allo stato "passivo" attraverso il metodo `ejbPassivate`. Tipicamente il container utilizza un algoritmo least-recently-used per decidere quale bean passare in stato "'"passivo". Se viene richiamato un metodo di un bean che si trova allo stato "passivo", allora il container provvederà prima a riattivarlo richiamando il metodo `ejbActivate` facendolo così ritornare allo stato "pronto", e poi a richiamare il metodo.

Prima di terminare la sessione il client invoca il metodo `remove` della home interface e il container, dopo averlo intercettato, provvede a deallocare il bean dalla memoria richiamando il metodo `ejbRemove`.

Lo *stateless session bean* non mantiene lo stato conversazionale del client. Quando un client invoca un metodo di uno stateless session bean, le sue variabili di istanza possono contenere uno stato, ma solo per la durata della invocazione. Tranne che durante l'invocazione di un metodo tutte le istanze del bean sono

equivalenti, lasciando al container il compito di assegnare un'istanza ad ogni client. La home interface di questo tipo di bean può contenere solo un metodo create senza argomenti.

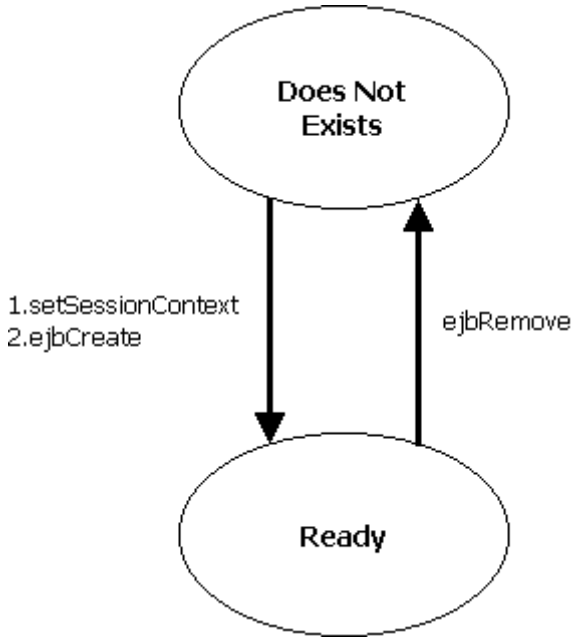


Figura 14-Ciclo di vita di uno stateless session bean

La figura 14 mostra il ciclo di vita di uno stateless session bean. Come si nota non esiste lo stato passivo in quanto non vi è necessità di salvare questo tipo di bean in memoria persistente, perché le risorse occupate devono essere mantenute per un periodo di tempo molto più breve rispetto al stateful session bean.
Inoltre il client non ha accesso ad un metodo `remove` poiché la rimozione del bean avviene solo da parte del container.

Entity Beans

Un entity bean differisce da un session bean per i seguenti aspetti:

- è persistente, nel senso che viene mantenuto lo stato su un database anche al di fuori del ciclo di vita della applicazione e dei processi del J2EE middle server. Ci sono due possibili tipi di persistenza che possono essere dichiarati nel deployment descriptor: *bean-menaged-persistence (BMP)* e *container-menaged-persistence (CMP)*. La prima impone allo sviluppatore di prevedere nel codice tutte le chiamate al database con opportuni statement SQL necessari al container per gestire il ciclo di vita del bean; ad esempio il metodo ejbCreate richiamato dal container invierà lo statement SQL di insert opportunamente fornito dallo sviluppatore. La seconda consente invece allo sviluppatore di non curarsi delle chiamate al database, delle quali si occupa automaticamente il container senza necessità di includere alcun statement SQL nel codice.

- consente accessi concorrenti da parte di più client. A tal scopo è importante che si utilizzino entity beans sempre all'interno di transazioni, che possono essere gestite automaticamente dal sistema.
- ha un identificatore unico (primary key).

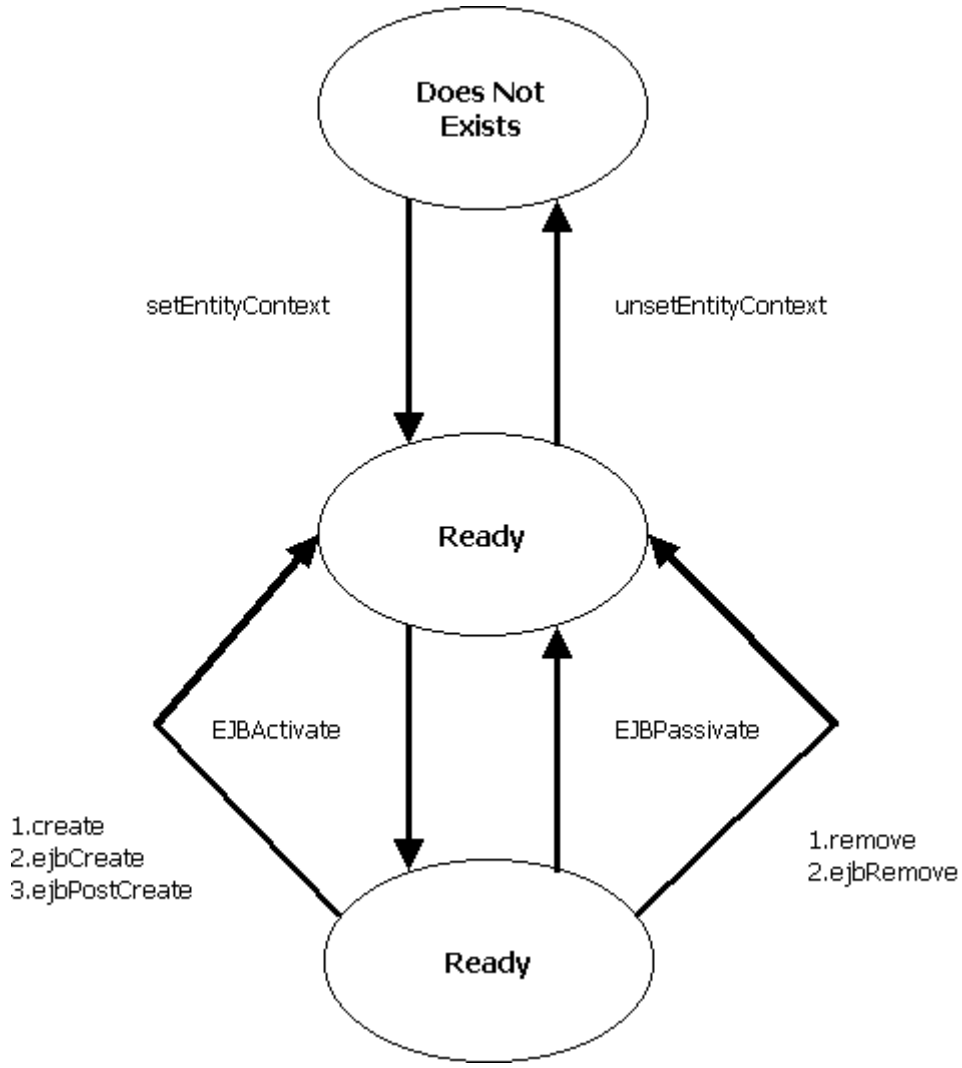


Figura 15-Ciclo di vita di un entity bean

La figura 15 riporta il ciclo di vita di un entity bean. Nel momento della partenza del server, tutti gli entity bean del quale è stato fatto il deployment vengono caricati in memoria in un pool di entity bean, che si trovano dunque in uno stato "pooled". In questo stato tutte le variabili assumono il loro valore di default ed il container assegna l'entity context col metodo `setEntityContext`. A questo punto ci sono due modi possibili in cui il bean può passare dallo stato "pooled" a quello "pronto". Il primo si ha quando il bean viene creato dal client con l'invocazione del metodo create: in questo caso in realtà non viene creato, ma viene prelevato un bean disponibile dal pool ed assegnato al client; successivamente il container richiama il

metodo `ejbCreate` corrispondente e il metodo `ejbPostCreate` per terminare la fase di inizializzazione. Il secondo modo si ha quando il container invoca il metodo `ejbActvate` per recuperare un bean che era stato deallocato dalla memoria primaria. Esistono anche due percorsi che portano dallo stato "pronto" allo stato "pooled" e sono per invocazione del metodo `ejbPassivate` da parte del container o del metodo `remove` da parte del client.
Alla fine del ciclo di vita di un entity bean il container richiama il metodo `unsetEntityContext` che rimuove il bean dal pool.
Come si è detto ogni entity bean ha un suo identificatore unico, che ne costituisce la primary key. Questa può essere implementata da una qualsiasi classe Java serializzabile. Tipicamente questa classe contiene semplicemente delle variabili di istanza corrispondenti alle variabili di istanza della primary key della tabella del database associata. Il client può utilizzare il metodo `FindByPrimaryKey`, che deve essere implementato nella classe del bean, passando come argomento una particolare istanza della classe primari key per ottenere l'entity bean specifico corrispondente. Il client può avere a disposizione anche altri metodi cosiddetti *finder* per recuperare una particolare entity bean passando altri valori univoci.
Il legame fra un entity bean e i dati che rappresenta sul database è talmente stretto che una modifica nel primo si riflette in un aggiornamento dei dati. Se il database da utilizzare per la memorizzazione dei dati è di tipo relazionale, però, si può creare un problema nel mappaggio dei dati con le variabili del bean. Infatti la logica a oggetti del linguaggio Java consente di realizzare strutture dati di complessità elevata rendendo difficile l'operazione di conversione nel formato accettato da un database relazionale. Il container ha a disposizione due metodi, `ejbLoad` e `ejbStore`, per sincronizzare i dati memorizzati nel database con quelli incapsulati nel bean. Tipicamente ogni invocazione di un metodo associato ad una transazione prevede prima l'invocazione di `ejbLoad`, poi l'esecuzione del metodo ed infine l'invocazione di `ejbStore`.

2.4.2 Un esempio di EJB

Si riporta a titolo di esempio il codice di uno stateless session bean che realizza una calcolatrice che effettua due semplici operazioni: il quadrato e il cubo di un numero.

Codice dell'EJB

Si è supposto di includere le classi Java dell'EJB in un package di nome "ejb".
Per prima cosa si scrive il codice della remote interface che contiene la dichiarazione dei metodi che potranno essere invocati dal client.

```
Package ejb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Calcolatrice extends EJBObject {
```

```
      public double Quadrato (double numero) throws
RemoteException;

      public double Cubo (double numero) throws
RemoteException;

}
```

Remote interface dell'EJB

Il passo successivo è quello di scrivere il codice della home interface che contiene i metodi di creazione e rimozione del bean. Essendo stateless non occorre alcun metodo di rimozione, ma solo di creazione.

```
Package ejb;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CalcolatriceHome extends EJBHome {

      Calcolatrice create () throws RemoteException,
CreateException;

}
```

Home interface dell'EJB

Poi si scrive il codice della classe del bean che implementa i metodi della remote interface.

```
Package ejb;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext ;

public class CalcolatriceEJB implements SessionBean {

       public double Quadrato (double numero) {
            return numero*numero ;
```

```
        }

        public double Cubo (double numero) {
             return numero*numero*numero;
        }

        public CalcolatriceEJB () {}
        public void ejbCreate () {}
        public void ejbRemove () {}
        public void ejbActivate () {}
        public void ejbPassivate () {}
        public void setSessionContext (SessionContext sc) {}
}
```

Classe dell'EJB

Il deployment descriptor dell'EJB

Una volta compilate le interfacce e la classe del bean, è necessario farne il deployment impacchettandole in un file EJB-JAR assieme al deployment descriptor.
Il deployment descriptor è un file XML, di cui si è già accennato e che deve chiamarsi ejb-jar.xml, che contiene informazioni specifiche sul bean e consente all'EJB container di istanziarlo e mettere a sua disposizione i servizi per cui viene configurato. Le specifiche EJB forniscono il DTD che deve essere seguito per scrivere il deployment descriptor.
Per una trattazione generale sui deployment descriptors si veda il paragrafo 2.6.
Si riporta qui di seguito la configurazione minima dell'EJB della calcolatrice da riportare nel deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar>

<description>EJB che consente di eseguire il quadrato e il cubo di un
numero</description>

<display-name>Calcolatrice</ display-name >

<enterprise-beans>
<session>

      <ejb-name>ejb.Calcolatrice</ejb-name>
```

```
      <home> ejb.CalcolatriceHome</home>
      <remote> ejb.Calcolatrice</remote>
      <ejb-class> ejb.CalcolatriceEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
</session>
</enterprise-beans>

</ejb-jar>
```

Deployment descriptor

Come si nota il file deve sempre cominciare con un tag `<ejb-jar>` e al suo interno si trova un altro tag `<enterprise-beans>` che contiene a sua volta un tag `<session>` e/o un tag `<entity>` per ogni session bean e/o entity bean contenuto nel file .jar. Nell'esempio il pacchetto conterrà solo un session bean, perciò viene riportato un unico tag `<session>` che conterrà i parametri di configurazione del bean Calcolatrice. Tra questi quelli obbligatori sono il nome completo della remote e home interface, il tipo di session bean (stateless o stateful) e il tipo di gestione delle transazioni che si intende utilizzare. E' attraverso quest'ultimo parametro che viene data la possibilità agli sviluppatori di specificare se la gestione delle transazioni deve essere fatta automaticamente dal container (*Container-menaged transaction demarcation*) o esplicitamente inserita nel codice del bean (*Bean-Managed Transaction Demarcation*).
Nell'esempio si è specificato con transaction-type il valore "Bean" che significa che si lascia allo sviluppatore il compito di definire nel codice i confini delle transazioni.
Specificando invece il valore "Container" si opta per una gestione automatizzata delle transazioni e in tal caso è necessario specificare per ogni metodo del bean un ulteriore transaction-attribute il cui valore indica quale tipo di definizione dei confini delle transazioni si vuole applicata al metodo stesso. I possibili valori di questo attributo sono i seguenti:

1. *Required*: il container garantisce che il metodo sia invocato sempre nel contesto di una transazione JTA. Se il client chiamante è già associato ad una transazione JTA si mantien lo stesso contesto con meccanismo di propagazione automatica, altrimenti viene creata una transazione JTA che comincia all'inizio del metodo e fa commit al ritorno, con gestione automatica del rollback nel caso venga lanciata un'eccezione.
2. *RequiresNew*: il container crea sempre una nuova transazione JTA prima dell'invocazione del metodo e fa il commit al ritorno. Se il client chiamante è già associato ad una transazione JTA, questa viene temporaneamente sospesa e ripresa al ritorno.
3. *NotSupported*: il contesto transazionale del client non viene propagato al metodo invocato. Se il client chiamante è già associato ad una transazione JTA, questa viene temporaneamente sospesa e ripresa al ritorno.
4. *Supports*: se il client chiamante è già associato al contesto di una transazione JTA questo viene propagato al metodo, altrimenti il metodo viene eseguito senza associazione ad alcuna transazione.

5. *Mandatory*: il container richiede che il client chiamante sia già associato ad una transazione JTA altrimenti viene lanciata un'eccezione.
6. *Never*: il container richiede che il client chiamante non sia già associato ad una transazione JTA altrimenti viene lanciata un'eccezione.

Codice del client

Ora si passa ad illustrare i punti più significativi del codice di un client che intenda utilizzare l'EJB appena creato.
La prima cosa che un client deve fare per poter utilizzare un EJB è recuperare una istanza della sua home interface attraverso un meccanismo che consenta di rintracciare oggetti remoti in esecuzione in spazi di indirizzamento eterogenei. Per fare questo è necessario poter riferire l'oggetto in questione attraverso un nome logico che sia indipendente dallo spazio di indirizzamento remoto.
La J2EE utilizza come standard per consentire una associazione tra risorse e nomi logici le API Java JNDI di cui si è già accennato. Esse forniscono due tipi di servizi:

1. un servizio di *binding* cioè la registrazione di un oggetto tramite un nome logico;
2. un servizio di *lookup* cioè il recupero di un oggetto a partire dal suo nome logico.

Al momento del deployment il container effettua un'operazione di binding dell'home interface del bean sotto il nome logico, specificato nel tag `<ejb-name>` del deployment descriptor, relativo ad un certo *naming context* che la J2EE definisce con il nome "java:comp/env" e che funge da radice nella gerarchia dei nomi.
Il nome logico completo con il quale un client può riferire il bean della Calcolatrice è dunque "java:comp/env/ejb/Calcolatrice".
Le specifiche EJB raccomandano di ottenere la home interface di un EJB come Object generico attraverso il nome JNDIconun'operazione di lookup, e poi di convertirlo in un'istanza della home interface utilizzando il metodo `narrow` della classe `javax.rmi.PortableRemoteObject`.
Si riporta qui di seguito il codice di un semplicissimo client che ottiene una reference alla home interface del EJB Calcolatrice ed esegue i suoi metodi per alcuni valori stampandone il risultato.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import Calcolatrice;
import CalcolatriceHome;

public class CalcolatriceClient {

      public static void main ( String[] args ) {
      try {
            // si ricava il context iniziale
            Context initial = new InitialContext ();
```

```
            // poi si ottiene un riferimento alla home interface come oggetto
generico
            Object objref = initial.lookup ("java:comp/env/ejb/Calcolatrice");

            // si converte l'oggetto in una istanza della home interface
            CalcolatriceHome home =
(CalcolatriceHome)PortableRemoteObject.narrow ( objref,

CalcolatriceHome.class );

            Calcolatrice calc = home.create () ;
            double ris = calc.Quadrato (50);
            System.out.println (String.valueOf (ris) );
            ris = calc.Cubo (50);
            System.out.println (String.valueOf (ris) );
            Calc.remove ();

      } catch ( Exception ex ) {

            System.err.println ("Si è verificato un errore: ");
            ex.printStackTrace ();
      }
}
```

Un semplice client che utilizza l'EJB

2.4.3 Web tier

Esistono diverse tecnologie che consentono la generazione di pagine web dinamiche sulla base dei dati provenienti da varie sorgenti.
La prima tecnologia sviluppata era la CGI (*Common Gateway Interface*), un'interfaccia per mezzo della quale il server web era in grado di lanciare applicazioni residenti sul server stesso.
La J2EE platform raccomanda invece l'utilizzo di JSP e servlet per la generazione di pagine web dinamiche. Tali tecnologie consentono di sviluppare applicazioni web con caratteristiche di portabilità, scalabilità e mantenibilità non raggiungibili attraverso la CGI.
La CGI infatti impone alcune limitazioni:

1. il codice inserito in un CGI script che accede a risorse come il file system o un database deve essere specifico per la piattaforma del server, perciò viene a mancare il vantaggio della portabilità che non

consente l'utilizzo di questa tecnologia in ambienti distribuiti dove la applicazioni web devono funzionare su piattaforme diverse;
2. ogni volta che uno script CGI viene invocato deve essere creato un nuovo processo, con conseguente spreco di risorse, lentezza d'esecuzione e limitata scalabilità;
3. le applicazioni che utilizzano CGI mischiamo il codice relativo alla presentation-logice e quello relativo alla business logic.

Servlet

Le servlet sono componenti web scritti in linguaggio Java. Tali componenti possono dunque essere supportati da web-server con qualsiasi piattaforma e possono essere portati su altre piattaforme senza necessità di essere ricompilati.
Inoltre le servlet vengono caricate in memoria una sola volta e ogni volta che una http request le richiede, questa viene servita da un thread diverso della stessa JVM, invece che da un nuovo processo, perciò il vantaggio della scalabilità è assicurato senza necessità di hardware aggiuntivo.
Infine uno dei benefici maggiori che merita di essere menzionato è la possibilità di utilizzare API uniformi per mantenere dati visibili attraverso richieste HTTP multiple, con particolare riferimento al concetto di sessione.
A proposito di quest'ultima caratteristica è bene tener presente che esistono quattro ambiti (*scope*) di esistenza, condivisione e visibilità delle sorgenti di informazione di una applicazione web; ciascuno di questi scope è associato ad un particolare oggetto, detto appunto *scope objetc*, e le informazioni ad esso associate vengono mantenute come attributi di tale oggetto.
I quattro scope object con la relativa visibilità sono i seguenti:

1. *web context*: rappresenta l'ambito con maggiore visibilità; la informazioni memorizzate come attributi sono visibili a tutti i web component dell'applicazione;
2. *session*: rappresenta un ambito associato ad un utente; le informazioni memorizzate come suoi attributi sono visibili a tutti i web component che servono le request di uno stesso client. E' questo il classico esempio del carrello della spesa delle applicazioni e-commerce;
3. *request*: rappresenta l'ambito associato ad una stessa request;
4. *page*: rappresenta l'ambito associato ad una stessa JSP.

Il problema di avere risorse condivise comporta una adeguata gestione della concorrenza tra i vari accessi. Tra gli scenari possibili di accessi concorrenti non vi sono solo quelli in cui più web components accedono a risorse del web context, o della stessa session, ma anche quelli in cui più thread dello stesso web component accedono a variabili di istanza. Come si è detto infatti il web container crea un nuovo thread per ciascuna request e non è quindi garantita l'atomicità dell'esecuzione di un metodo, a meno di non specificare la clausola synchronized. Un altro sistema per garantire l'atomicità del metodo service (quello che serve di fatto la request) è quello di far implementare alla servlet l'interfaccia `SingleThreadModel`.
E' opportuno dichiarare synchronized tutti i metodi che interagiscono con risorse condivise, siano esse residenti in memoria centrale o secondaria (file system e databases).

Le servlet hanno un ciclo di vita che viene gestito dal web-container. Quando una request è mappata ad una servlet, il container esegue i seguenti passi:

1. carica la classe Java della servlet;
2. ne crea una istanza;
3. inizializza l'istanza invocando il metodo `init`; questo metodo viene richiamato una sola volta all'atto del caricamento della servlet stessa e può essere ridefinito per personalizzare la configurazione iniziale dell'istanza. Viene inoltre assicurata la sincronizzazione automatica dell'esecuzione di questo metodo in modo da consentirne l'accesso ad un thread per volta;
4. invoca il metodo service passando come parametri un oggetto `HttpRequest` e un oggetto `HttpResponse`.

Se il container necessita di rimuovere una servlet ne richiama il metodo `destroy`.
Volendo entrare un po' più nei dettagli si riporta qui di seguito un breve esempio di una servlet.

```
import java.util.*;
import java.text.*;

public class SimpleServlet extends HttpServlet {

    // ridefinizione del metodo doGet

        public void doGet ( HttpServletRequest request, HttpServletResponse respose )
                    throws Servlet Exception, IOException {
            PrintWriter out;

            // istanziamento degli oggetti per data e ora
            String dateformat = "EEEE d MMMM yyyy";
            String timeformat = "H:mm";
            DateFormat df = new SimpleDateFormat ( dateformat );
            DateFormat tf = new SimpleDateFormat ( timeformat );
            Date datetime = new Date ();

            // prima occorre settare il content type
            response.setContentType ("text/html");

            // poi costruire la response
            out = response.getWriter ();
            out.println ("<HTML><HEAD><TITLE>");
            out.println ("DATA E ORA");
            out.println ("</TITLE></HEAD><BODY>");
```

```
                out.println ("<H1>DATA E ORA ATTUALI</H1>");

                out.println ( df.format ( datetime ) );
                out.println ("<BR>");
                out.println ( tf.format ( datetime ) );
                out.println ("</BODY></HTML>");
                out.close ();
        }
}
```

Esempio di una servlet che genera una pagina dinamica con data e ora

Questo è un banalissimo esempio di una servlet che genera una pagina web che contiene la data e l'ora attuali nel momento in cui la pagina viene spedita al client.
Si analizzano ora brevemente le parti più significative del codice.
Una classe Java per essere una servlet deve estendere la classe astratta `HttpServlet` e deve ridefinire almeno un metodo di questa classe tra i seguenti: `doGet`, `doPost`, `doDelete`, `doPut` a seconda del tipo di request HTTP da gestire. In particolare il metodo `doGet` gestisce le request HTTP di tipo GET, cioè quelle che includono eventuali parametri attaccandoli in coda all'URL consentendo quindi un limitato invio di dati, mentre il metodo `doPost` gestisce le request HTTP di tipo POST che includono eventuali parametri nella request stessa, consentendo l'invio di un quantitativo di dati molto più ampio. A tutti questi metodi devono essere passati come argomenti la `HttpRequest` ricevuta dal client e la `HttpResponse` da settare ed inviare al client.
Nell'esempio il metodo utilizzato è il `doGet` e la request non contiene parametri. Inoltre si è fatto uso di istanze di classi apposite per ottenere la data attuale opportunamente formattata.
Per costruire la response occorre prima settare i campi dell'intestazione (*header fields*): tra questi in particolare il campo content-type è quello che definisce il tipo di contenuto del body della response. Nell'esempio il metodo `setContentType` dell'interfaccia `HttpResponse` viene utilizzato per settare il content-type al valore "text/html" che indica che il body della response conterrà del testo in formato HTML, cioè la sorgente di una pagina web che può essere interpretata da un browser.
Per costruire il body della response occorre innanzitutto ottenere un output strema da usare per inviare dati al client. Dal momento che, come si è detto, il contenuto della response dovrà essere interpretato dal browser del client come testo HTML, nell'esempio si è utilizzato il metodo `getWriter` per ottenere un oggetto `PrintWriter` che consente di inviare uno strema di caratteri. Se si fosse utilizzato il metodo `getOutputStream` si sarebbe ottenuto un oggetto `OutputStream` associato alla response sul quale sarebbe stato possibile inviare un generico stream di bytes in forma binaria (è questa la tecnica adottata per eseguire il download dei files nell'applicazione realizzata).

Come si nota le servlet utilizzano degli statement Java di stampa per l'invio di HTML al client. La pagina viene costruita dal codice dell'applicazione che scrive nello stream di output il codice HTML carattere per carattere.
Il risultato dell'elaborazione di questa servlet è la seguente pagina HTML:

```
<HTML>
      <HEAD>
            <TITLE>DATA E ORA</TITLE>
      </HEAD>
      <BODY>
            <H1>DATA E ORA ATTUALI</H1>
            lunedì 4 febbraio 2002<BR>
              15:58
      </BODY>
</HTML>
```

HTML prodotto dalla servlet precedente

JSP (Java Server Pages)

Includere l'HTML negli statement di stampa, come avviene nelle servlets, presenta due inconvenienti:

1. un Web designer non riesce ad avere una chiara visione di come sarà l'aspetto della pagina generata prima che la servlet venga eseguita a run-time;
2. se si deve realizzare una modifica nell'aspetto della pagina è molto difficile localizzare la sezione di codice responsabile di tale aspetto.

L'esempio di servlet illustrato precedentemente è molto semplice, per cui non risultano molto evidenti gli inconvenienti di cui sopra. Si immagini però una servlet che generi una pagina HTML molto più elaborata con tantissimi tags innestati tra loro e si avrà così chiara la problematica in questione.
Per ovviare a questi problemi la J2EE raccomanda l'utilizzo di JSP per la generazione di pagine web dinamiche.
Le JSP sono documenti di testo che forniscono una descrizione di come processare una request per creare una response utilizzando un determinato protocollo (l'HTTP è quello di default).
Sarà compito del web-container interpretare nel modo corretto il contenuto di una JSP traducendolo in una servlet che potrà poi essere eseguita dal server per generare la pagina dinamica. Le specifiche servlet e JSP della Sun forniscono le direttive standard che deve seguire un web-container che supporta le servlet (detto quindi anche servlet-container).
Una JSP è quindi una combinazione di due tipologie di testo:

1. template fissi, che rappresentano il contenuto statico destinato a rimanere tale;

2. espressioni JSP basate sull'utilizzo di diversi paradigmi, che verranno tradotte opportunamente del web-container.

I template statici possono essere inseriti direttamente nella pagina senza doverli racchiudere in statement di stampa, cosa che verrà fatta automaticamente dal web-container. Inoltre il contenuto statico non si limita al solo HTML, ma può comprendere anche qualsiasi altro formato text-based come l'XML.

Per quanto riguarda le espressioni JSP, queste possono essere riconosciute come tali e opportunamente interpretate dal web-container solo se rientrano nelle seguenti quattro differenti categorie:

1. ***Directive***: sono direttive di carattere generale, indipendenti dal contenuto specifico della pagina, relative alla fase di traduzione/compilazione; Sintassi: `<%@ directive … %>`.

   Le directive sono tre:

   - *page*: definisce diverse proprietà relative alla pagina stessa che vengono comunicate al web-container;
   - *taglib*: dichiara le tag libraries che verranno usate nella pagina;
   - *include*: comunica al container che nella fase di traduzione deve includere nella pagina il contenuto di un file il cui nome viene specificato nella direttiva stessa;

2. ***Action***: vengono eseguite nella fase di processing e danno origine a codice Java specifico per la loro esecuzione; Sintassi: quella dei tags XML, cioè `< tag attr1="value1" attr2="value2" … > body </tag>`
3. ***Scripting element***: sono frammenti di codice Java, o in un altro linguaggio specificato e si dividono a loro volta in tre sottocategorie:

   - *Scriplet*: sono veri e propri statement scritti nello scripting language che danno origine a porzioni di codice Java all'atto della traduzione in servlet; Sintassi: `<% codice %>`
   - *Declaration*: sono dichiarazioni che vengono inserite nella classe Servlet come elementi della classe stessa, senza essere racchiuse in alcun metodo. Possono essere sia variabili di classe sia metodi. Se si tratta di variabili la loro durata è quella del Servlet, quindi sopravvivono e conservano il loro valore nel corso di tutte le esecuzioni dello stesso Servlet. Sintassi: `<% ! declaration [declaration] … %>`
   - *Expression*: contengono un'espressione che segue le regole delle espressioni dello scripting language; l'espressione viene valutata e scritta nella pagina di risposta nella posizione corrispondente a quella dell'espressione JSP. Sintassi: `<% = expression %>`
   - *Comment*: sono commenti che riguardano la pagina JSP e che vengono eliminati dal web-container nella fase di traduzione; sintassi: `<% -- comment -- %>`

Una JSP che realizza la pagina dell'esempio precedente è la seguente:

```
<%    @ page import = "java.util.*, java.text.*" contentType = "text/html" %>

<%    String dateformat = "EEEE d MMMM yyyy";
      String timeformat = "H:mm";
```

```
      DateFormat df = new SimpleDateFormat (dateformat);
      DateFormat tf = new SimpleDateFormat (timeformat);
      Date datetime = new Date ();
%>

<HTML>
      <HEAD>
            <TITLE>DATA E ORA</TITLE>
      </HEAD>
      <BODY>
            <H1>DATA E ORA ATTUALI</H1>
            <%= df.format (datetime) %><BR>
            <%= tf.format (datetime) %>
      </BODY>
</HTML>
```

JSP che realizza la pagina dinamica con data e ora

Essa verrà tradotta dal web-container nella servlet precedente la cui esecuzione fornisce esattamente la pagina HTML dell'esempio visto prima.

Si è fatto uso di una Directive page nella quale si sono specificate due proprietà associate alla JSP, la prima mediante l'attributo import al quale è stato assegnato come valore la lista dei package Java da importare, la seconda mediante l'attributo contentType al quale si è assegnato il valore "text/html", cioè il tipo di contenuto da inviare al client come pagina HTML. Quest'ultimo attributo se non specificato viene assunto per default come "text/html" con charset = Latin-1 ad indicare il set di caratteri utilizzato per codificare la pagina.

Come si può notare, per mezzo di una JSP un Web designer può avere subito una immagine chiara di come risulterà la pagina web. Inoltre, essendo le JSP dei documenti di testo, egli può anche utilizzare dei tool per creare e visionare il loro contenuto.

La J2EE raccomanda anche di fare meno uso possibile di elementi scriplet.

Una JSP con molti scriplet al suo interno, infatti, può causare alcuni problemi:

1. diventa più difficile da leggere ed è necessaria la conoscenza del linguaggio di scripting per capirne la funzionalità;
2. nel caso in cui alcune funzionalità debbano essere ripetute in più pagine, occorre riportare gli scriplet ricopiandoli con un copia e incolla, con evidente ridondanza di codice e difficoltà di manutenzione.

Al contrario una JSP con pochi scriplet, o addirittura senza, non solo evita i problemi sopraelencati, ma consente una ancor più netta separazione del codice dalla presentazione, favorendo in tal modo la manutenzione e la separazione dei ruoli, e quindi il parallelismo, tra il web designer che può concentrarsi sull'aspetto della presentazione, e il web developer che si occupa dei contenuti trascurando l'interfacciamento grafico.

I due principali meccanismi che consentono di raggiungere questo obiettivo sono i JavaBeans e i Custom Tags.

JavaBeans

I JavaBeans sono delle classi Java facilmente riusabili e composte insieme in una applicazione. Ogni classe Java che segue certe regole di composizione può essere un JavaBean component.
Tali regole governano le proprietà della classe del JavaBean e i metodi pubblici che ne danno accesso.
Ogni JavaBean, infatti, ha una serie di proprietà che possono essere di sola lettura, di sola scrittura o di lettura e scrittura.
Ogni proprietà leggibile deve avere un metodo pubblico getter del tipo

```
PropertyClass getProperty () { … }
```

E ogni proprietà settabile deve aver un metodo pubblico setter del tipo

```
SetProperty ( PropertyClass pc ) { … }
```

Inoltre un JavaBean deve avere un costruttore senza parametric.
La JSP supportano l'utilizzo di JavaBeans diretto attraverso Actions che possono essere inserite nella pagina:

1. ***jsp:UseBean***: serve per utilizzare un bean già esistente o crearne una nuova istanza;
   i suoi attributi sono:
   - *id*: definisce il nome di una variabile che identifica il bean nello scope specificato;
   - *scope*: definisce l'ambito di esistenza e di visibilità della variabile il cui nome è definito dall'attributo id. Si tratta dunque di un attributo che specifica a quale scope object andrà associato il bean. I valori ammessi sono i seguenti:
     i) *page*: la variabile è utilizzabile solo all'interno della pagina in cui compare il tag, o di una pagina inclusa staticamente;
     ii) *request*: la variabile è utilizzabile nell'ambito di una singola request;
     iii) *session*: la variabile è utilizzabile nell'ambito di un'intera sessione; la pagina in cui il bean è creato deve contenere una direttiva page con l'attributo session settato a true;
     iv) *application*: la variabile è utilizzabile nell'ambito dell'intera applicazione da tutte la sue pagine JSP.
   - *class*: definisce il nome della classe Java a cui il ben appartiene;
   - *type*: identifica il tipo della variabile il cui nome è definito dall'attributo id, che può così anche non essere necessariamente la classe specificata dall'attributo class, ma anche una superclasse o un'interfaccia implementata dalla classe;
2. ***jsp:getProperty***: inserisce nella pagina il valore di una proprietà del bean;
3. ***jsp:setProperty***: assegna il valore di una o più proprietà del bean.

Il body contenuto all'interno dell'Action `jsp:useBean` viene eseguito solo all'atto dell'istanziamento del bean.
L'utilizzo dei JavaBeans è quindi vantaggioso poiché consente di incapsulare del codice Java al suo interno e di inserirlo nella JSP attraverso i tag sopraelencati, invece di immetterlo direttamente nella pagina come scriplet.
Dalla pagina è possibile manipolare il bean attraverso il settaggio ed il ripescaggio delle sue property, ma anche richiamando direttamente da uno scripting element un suo metodo attraverso la variabile identificata col nome specificato nell'attributo id.
Ritornando all'esempio di prima, si può creare il seguente bean:

```
import java.util.* ;
import java.text.* ;

public class DateTime {

      String dateformat = "EEEE d MMMM yyyy";
      String timeformat = "H:mm";
      DateFormat df = new SimpleDateFormat (dateformat);
      DateFormat tf = new SimpleDateFormat (timeformat);

      Public String getDate () {

            Date datetime = new Date ();
            Return df.format (datetime);

      {

      public String getTime () {

            Date datetime = new Date ();
            Return tf.format (datetime);

      }
}
```

JavaBean per incapsulare il codice per visualizzare data e ora

Poi si può richiamare il bean dalla pagina utilizzando le Actions descritte, perciò la JSP avrà il formato riportato qui di seguito, nel quale si è omessa la Directive non essendo più necessario specificare l'attributo import e non avendo assunto il contentType di default:

```
<jsp:useBean id="datetime" class="DateTime" scope="page"/>

<HTML>
      <HEAD>
            < TITLE >DATA E ORA</TITLE>
      </HEAD>
      <BODY>
            <H1>DATA E ORA ATTUALI</H1>
            <JSP:getProperty name="datetime" property="date"/>
            <BR>
            <jsp: getProperty name="datetime" property="time"/>
      </BODY>
</HTML>
```

JSP che utilizza il precedente JavaBean per generare data e ora

In quest'ultima versione della pagina JSP si può apprezzare la differenza con la versione script: facendo uso del JavaBean per incapsulare il codice Java la pagina appare molto più pulita e compatta e il contenuto risulta di facile e intuitiva comprensione. Inoltre tutta la funzionalità racchiusa nel JavaBean può essere utilizzata anche da web designers che, senza necessità di conoscenza del linguaggio Java, possono occuparsi esclusivamente del layout delle pagine in cui dovrà essere inserita la data e l'ora. Qualora si volesse effettuare una qualunque modifica, ad esempio cambiare il formato di visualizzazione della data, sarebbe sufficiente cambiare la stringa `dateformat` e ricompilare il JavaBean per ottenere automaticamente l'aggiornamento di tutte le pagine JSP che lo utilizzano, oppure si potrebbe prevedere un metodo `setFormat` che consenta di personalizzare il formato di pagina in pagina semplicemente settando una property del bean.

Custom Tags

Come si è visto i JavaBeans forniscono un potente strumento per incapsulare funzionalità di presentation logic da inserire in una JSP. Essi però sono manipolabili solo attraverso le loro properties, mentre in taluni casi potrebbe essere utile poter far uso di strumenti in grado di offrire maggiore flessibilità.
Dalla versione 1.1 delle specifiche JSP è stata introdotta un'importante innovazione: la definizione delle *tag extension*, ossia un meccanismo attraverso il quale gli sviluppatori possono estendere le JSP con tags creati ad hoc, i cosiddetti *custom tags*, che, una volta dichiarati e inclusi nella JSP, diventeranno delle action a tutti gli effetti. Possono così essere create delle vere e proprie librerie di nuovi tags (*tag libraries*) con le caratteristiche di ricusabilità e portabilità, che forniscono agli web designers potenti strumenti con sintassi XML a loro molto più familiari che non i linguaggi di scripting.

Per implementare un custom tag di deve definire una classe, chiamata genericamente *tag handler*, che implementi una delle tre interfacce `Tag`, `BodyTag` o `IterationTag`, definite nel package `javax.servlet.jsp.tagext`.

L'interfaccia `Tag` viene utilizzata per implementare un tipo di tag empty, cioè non dotato di body, oppure per implementare una action che richiede semplicemente l'esecuzione di operazioni quando viene incontrato quello finale. Questa interfaccia dunque contiene i metodi di base che sono tipici di tutti i tag handlers, ossia metodi setter per inizializzare il tag con un context e gli eventuali attributi dell'azione, e i metodi che devono essere ridefiniti per implementare le funzionalità vere e proprie da attribuire al custom tag; in particolare questi ultimi sono due: `doStartTag` e `doEndTag`.

Il metodo `doStartTag` deve contenere le operazioni che il container dovrà eseguire nel momento in cui nell'elaborazione della JSP viene incontrato il tag iniziale. Esso restituisce un valore intero che indica al container se e come compiere un processing del body, qualora il tag lo preveda. In particolare esistono due differenti valori di ritorno che indicano due differenti alternative:

1. ignorare il body (utilizzato per i tag empty);
2. valutare il body e inserirne la valutazione direttamente nello stream di output della response (utilizzato ad esempio per inclusioni condizionali).

Il metodo `doEndTag` deve contenere le operazioni che il container dovrà eseguire nel momento in cui nell'elaborazione della JSP viene incontrato il tag finale. Il suo valore di ritorno è un intero che indica al container se il resto della pagina deve essere valutato o meno.

L'interfaccia IterationTag è un'estensione dell'interfaccia Tag che fornisce un ulteriore metodo, `doAfterBody`, invocato dal container per la rivalutazione del body del tag, qualora si voglia implementare un tag iterativo. Esso infatti restituisce un intero che indica se il body deve essere rivalutato ancora o se deve essere richiamato il metodo `doEndTag`.

L'interfaccia `BodyTag` è un'estensione di `IterationTag` che contiene ulteriori due metodi che devono essere ridefiniti qualora si richieda una gestione più complessa del body del tag. Tali metodi sono: `setBodyContent`, `doInitBody`. Il primo è chiamato dal container prima della valutazione del body, solo nel caso questo abbia luogo, e assegna al tag un oggetto `BodyContent` che funge da buffer temporaneo per l'output del body. Il secondo viene chiamato dopo il primo soltanto alla prima valutazione del body (e non alle eventuali successive iterazioni).

Per l'interfaccia `BodyTag` il metodo doStartTag può ritornare, come avveniva per l'interfaccia Tag, un valore che indica al container di ignorare il body, mentre l'altra possibilità prevista è quella di valutare il body e inserirlo nel `BodyContent`, invece di mandarlo direttamente sullo stream di output.

Esistono anche delle classi predefinite che implementano le interfacce di cui sopra, e che possono essere utilizzate direttamente come tag handler, al limite ridefinendo solo alcuni metodi strettamente necessari. Esse sono `TagSupport` e `BodyTagSupport`.

All'interno di una JSP è possibile importare più librerie di tag specificandole attraverso la directive taglib. Ciò viene fatto riportando un attributo della directive il cui valore è un URI univocamente associato ad una tag library, e un ulteriore attributo il cui valore è il prefisso che verrà utilizzato per identificare i custom tags della tag library quando verranno inseriti nella JSP.

Nel deployment descriptor web.xml dell'applicazione web in cui la tag library viene utilizzata deve essere riportato un mapping tra l'URI della tag library e un percorso in cui si trova un particolare file XML detto TLD (*Tag Library Descriptor*).

```
<taglib>
      <taglib-uri>
            http://sparc.ing.unimo.it:8063/md/taglib
      </taglib-uri>

            <taglib-location>
                  /WEB-INF/taglib/tagdescriptor.tld
            </taglib-location>
</taglib>
```

Esempio della sezione del deployment descriptor che specifica una taglibrary

Il TLD è un file di configurazione in formato XML, che fornisce al container le informazioni necessarie per poter utilizzare correttamente la libreria. Esso riporta, per ciascuna action della tag library, il nome del tag, il nome della classe del tag handler, informazioni su tutte le eventuali variabili di scripting create dall'action e informazioni sugli eventuali attributi dell'action.
Questo file va inserito in un percorso dell'applicazione web che deve corrispondere a quello riportato nel tag `<taglib-location>`. Inoltre occorre inserire nel percorso in cui vengono tipicamente riportate le librerie di classi utilizzate dall'applicazione web, tutte le classi utilizzate nella tag library. Solitamente queste classi si trovano impacchettate in un file JAR da inserire nella directory WEB-INF/lib.
Il DTD del file TLD è riportato nelle specifiche JSP dalla versione 1.1 in poi.

2.5 Client tier

Costituisce il front-end più o meno complesso che risiede e viene eseguito sul client che accede all'applicazione. Esso inoltra le richieste al server per conto dell'utente e ne presenta a quest'ultimo i risultati.
La J2EE sopporta vari tipi di client che possono connettersi sia all'interno del firewall aziendale che da fuori attraverso il World Wide Web.
Un client può comunicare con e usare i servizi forniti da uno o più tiers dell'applicazione enterprise.
A seconda del tier a cui si collega si possono distinguere tre tipologie principali di client: il *Web-client, l'EJB-client e l'EIS-client*.

2.5.1 Web-client

Il web client solitamente viene eseguito all'interno di un browser e utilizza i servizi di quest'ultimo per interpretare il contenuto delle informazioni provenienti dal server. L'interfaccia utente è quindi generata a lato server dal web tier e comunicata via HTML.
Il web-browser è l'unico requisito di ambiente richiesto per questa tipologia di client, che, per questo motivo, viene anche detto "thin client" (client sottile).
A volte è utile fornire direttamente al client alcune funzionalità che aiutano l'utente a meglio interagire con i servizi del server. In tal caso oltre alle pagine HTML possono venire inviati anche dei Java Applets, cioè delle classi Java che vengono scaricate assieme alla pagina HTML che le riferisce ed esegue sul client dall'ambiente di runtime java del browser. Naturalmente le applets sono soggette a particolari restrizioni per garantire la sicurezza e la protezione del client.
La piattaforma J2EE fornisce un supporto speciale per il downloading e l'installazione automatica di una specifica JVM sul client.
I browser spesso supportano anche altri embedded components, come i plug-ins di Netscape e gli ActiveX di Internet Explorer che possono essere scaricati.
I Web-client utilizzano i protocolli di trasmissione HTTP e HTTPS che forniscono i seguenti vantaggi:

- sono diffusissimi: quasi tutti i computer attualmente hanno un browser che supporta l'HTTP;
- presentano il carattere della robustezza e semplicità;
- passano attraverso i fire-wall.

Tali protocolli presentano però allo stesso tempo anche alcuni svantaggi:

- sono stateless: l'HTTP è basato su un modello request-response nel quale a ciascuna richiesta fa corrispondere una e una sola risposta senza mantenere informazioni sulle precedenti richieste e risposte;
- sono non-transazionali: l'HTTP non supporta la possibilità di definire transazioni attraverso richieste multiple.

2.5.2 EJB-client

L'EJB-client è un application client che interagisce con l'EJB tier. Esso consente all'utente interazioni col server molto più potenti di un semplice Web-client grazie alla presenza di un'interfaccia utente più coplessa, tipicamente un programma GUI (*Graphic User Interface*), che deve quindi essere installata sul client stesso (il cui ambiente è solitamente un desktop computer).
Ciò presenta anche il vantaggio di avere una distribuzione maggiore del carico di lavoro tra client e server. In particolare il compito di generare l'interfaccia grafica è riservato al client, mentre al server viene lasciato il compito di implementare la sola business-logic. Questo è particolarmente utile per applicazioni con specifici requisiti grafici, per la quali sarebbe troppo gravoso lasciare al server tutta l'elaborazione.
Sono disponibili diversi servizi del middle-tier a questo tipo di client: JNDI, JMS, JDBC.
Per poter far uso di tali servizi il client deve essere eseguito in un container che tipicamente è molto più semplice dei container del middle-tier. Si tratta infatti di una libreria che viene distribuita assieme al

programma client. Essa è specifica per il particolare J2EE service provider, che generalmente provvede a fornirla sottoforma di pacchetto JAR ed un deployment descriptor che definisce gli EJB e le altre risorse esterne alle quali il client può accedere.
Non è richiesto che gli EJB-client supportino le transazioni (se nono attraverso gli strumenti forniti dalle API JDBC), perciò solitamente la loro gestione avviene attraverso gli EJB.
Per quanto riguarda la sicurezza esistono particolari restrizioni, in quanto secondo la J2EE platform l'EJB-client deve sempre essere autenticato per poter accedere al middle-tier. Le tecniche di autenticazione sono fornite dal client container e non sono direttamente controllabili dal client stesso.
Il protocollo di trasmissione utilizzato da questi client è l'RMI-IIOP, che consente l'accesso ad oggetti Java definiti usando l'interfaccia RMI (Remote Model Interface) attraverso il protocollo IIOP. Tale protocollo non è solitamente in grado di passare attraverso un fire-wall, se non con particolari setup aggiuntivi, e questo limita fortemente l'utilizzo di Internet da parte di questi client.

2.5.3 EIS-client

L'EIS-client accede direttamente alle risorse dell'Information System e si assume la responsabilità di mantenere il rispetto dei vincoli e regole della business logic dell'applicazione. Esso può utilizzare le API JDBC per accedere ai database relazionali, ma attualmente la J2EE non ha ancora definito uno standard implementativi.
Per il fatto che si deve occupare completamente sia dell'interfaccia utente che della business logic applicativa, questo tipo di client è scarsamente utilizzato e il suo utilizzo dovrebbe essere riservato a compiti di gestione e mantenimento per i quali è richiesta un'interfaccia minima o non esistente, come ad esempio uno stand-alone program di manutenzione delle tabelle di database relazionali da lanciare durante la notte.

2.6 Packaging e deployment

La J2EE platform consente agli sviluppatori di creare differenti parti delle loro applicazioni sotto forma di componenti riusabili. Questi componenti possono essere poi raggruppati in insiemi, detti *moduli*, a seconda di caratteristiche o funzionalità comuni. Il processo di assemblaggio dei componenti in moduli e dei moduli in enterprise applications è detto *packaging*.
Il processo di installazione e personalizzazione di un'applicazione in un certo ambiente operativo è detto *deployment*. Affinché sia possibile personalizzare un'applicazione i suoi componenti devono poter essere configurabili mediante un meccanismo standard.
La J2EE fornisce strumenti standard per semplificare questi processi di packaging e deployment. Essa usa file JAR come package standard per i moduli e le applicazioni, e dei files XML-based per configurare questi moduli detti *deployment descriptors*.

2.6.1 Ruoli e compiti

I processi di packaging e deployment coinvolgono tre differenti ruoli identificabili in tre figure professionali non necessariamente svolte da persone diverse:

1. *Application component providers*: hanno il compito di sviluppare gli Enterprise Java Beans, le pagine HTML e JSP, e tutte le ulteriori classi associate. Essi forniscono tutte le informazioni strutturali del deployment descriptors per ciascun componente, che includono, come già visto, l'indicazione della home e della remote interface e della classe degli EJB, il meccanismo di persistenza usato, e il tipo di risorse che questi componenti utilizzano. Senza l'ausilio dei deployment descriptors tutte queste informazioni sarebbero da includere nel codice dei componenti riducendo drasticamente la loro flessibilità e riusabilità.
2. *Application assemblers*: forniscono informazioni relative all'applicazione nel suo complesso. Tipicamente ad essi spetta il compito di mappare le varie servlet sui diversi URL, definire le pagine di errore da usare, i vincoli di sicurezza dei vari componenti, ecc.
3. *Deployers*: sono responsabili del deployment dei componenti in un particolare ambiente operativo. Ad essi spetta il compito di installare i diversi componenti sul J2EE server (o sui J2EE servers) fornendo le informazioni aggiuntive specifiche per l'ambiente operativo. Ad esempio essi si occupano di mappare gli utenti e gli accounts esistenti con i security roles definiti dall'Application assembler.

2.6.2 Moduli J2EE

Un'applicazione J2EE è impacchettata in un file Enterprise Archive (EAR), un file standard Java JAR con estensione .ear. L'obiettivo di questo meccanismo di packaging è di fornire una unità di deployment che sia portabile.

Un file EAR contiene uno o più moduli J2EE e un *J2EE application deployment descriptor*, che contiene indicazioni sui moduli stessi.

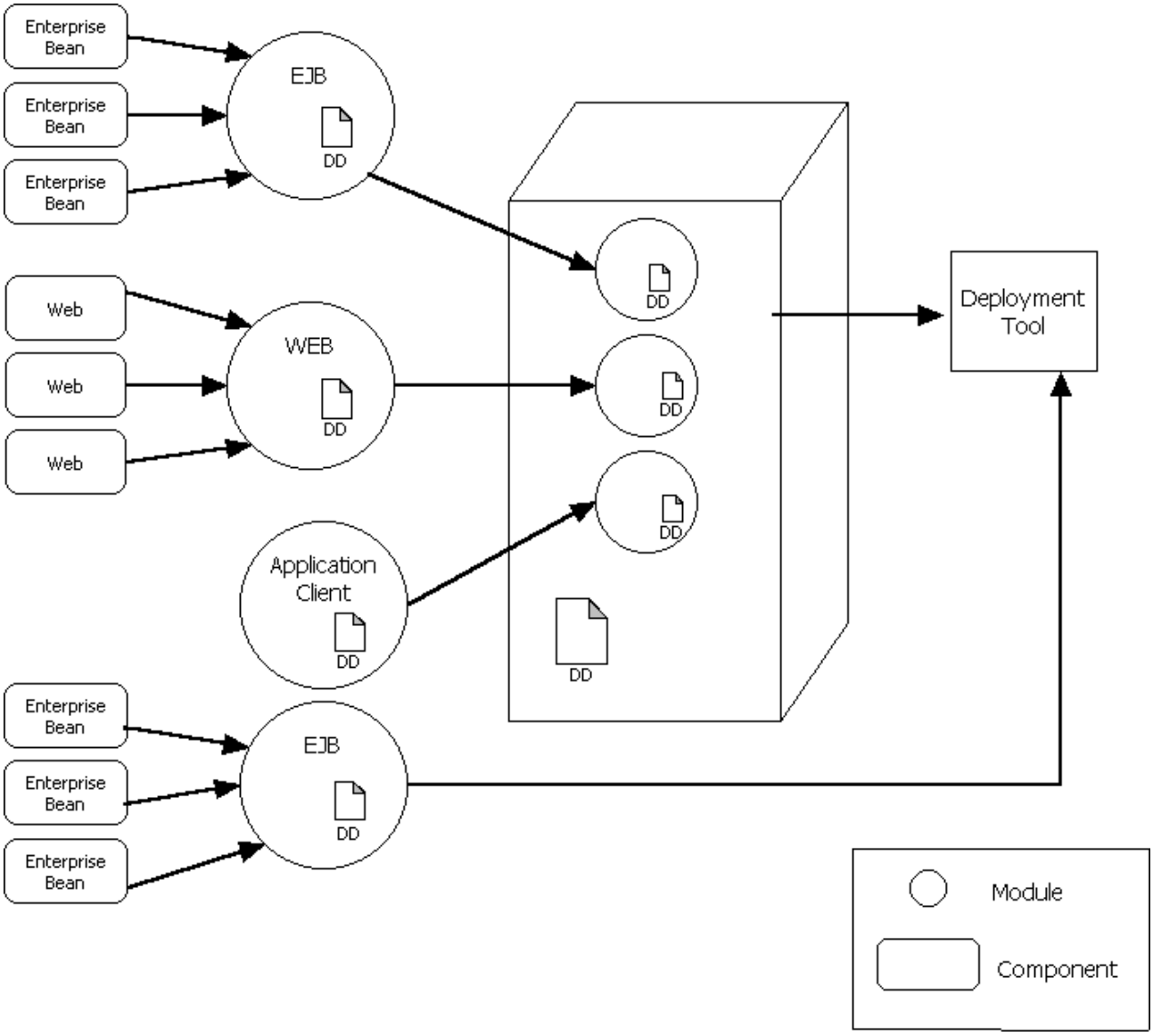


Figura 16-Packaging dei moduli J2EE

Moduli EJB

Un modulo EJB è l'unità più piccola di EJB della quale si può fare il deployment. Tale modulo è impacchettato in un file EJB JAR, cioè un file standard Java JAR con estensione .jar che contiene:

1. le classi Java degli EJB e le loro remote e home interface. Se si tratta di un entity bean deve essere compresa anche la classe della primary key;
2. tutte le altre classi Java dalle quali dipendono gli EJB che non siano già comprese nella J2EE platform;
3. un EJB deployment descriptors, tipicamente chiamato ejb-jar.xml, del quale si è già mostrato il contenuto minimo nell'esempio della Calcolatrice, che deve essere contenuto in una particolare directory di nome META-INF.

E' da notare che un EJB JAR file differisce da un normale JAR file perché nel primo è contenuto anche il deployment descriptors.

Moduli web

Un modulo Web è l'unità più piccola di risorse Web della quale si può fare il deployment. Tale modulo è impacchettato in un file Web Archive (WAR), un file standard Java JAR con estensione .war che contiene:

1. le classi Java delle servlet e le classi dalle quali queste dipendono, eventualmente impacchettate a loro volta in un normale file JAR;
2. le pagine JSP e le classi Java dalle quali dipendono;
3. documenti statici, come ad esempio le pagine HTML, le immagini, i file sonori, ecc.;
4. le applets;
5. un Web deployment descriptor, tipicamente chiamato web.xml e contenuto in una speciale directory chiamata WEB-INF.

Moduli Application client

Un modulo application client è impacchettato in un file JAR con estensione .jar e contiene:

1. le classi Java che implementano il client;
2. un Application client deployment descriptor che descrive gli EJB e le risorse esterne referenziate dall'applicazione.

La J2EE non specifica tools per effettuare il deployment di un application client, o meccanismi per installarlo. Alcune piattaforme J2EE sofisticate possono consentire il deployment dell'application client direttamente nel J2EE server e metterlo così automaticamente a disposizione di alcuni clients intranet. Altre piattaforme J2EE possono invece richiedere che il deployment dell'application client sia manualmente effettuato su ciascuna macchina client.

2.7 Connection pooling

Quando un componente intende accedere a risorse di un database deve stabilire una connessione. I tempi di attesa per stabilire una connessione sono molto onerosi e possono compromettere le prestazioni di un middle tier server.

Per ovviare a questo inconveniente la J2EE fornisce supporto per il *connection pooling*, cioè si occupa di mantenere e gestire una cache di connessioni al database, così da consentirne il riuso.

Questo supporto viene fornito dall'EJB container in modo del tutto trasparente agli sviluppatori che nel codice dei componenti richiedono una connessione e la richiudono una volta terminata. Sarà compito del container assegnare al componente una delle connessioni già aperte e disponibili nella cache, apporvi un lock in modo che non sia utilizzabile da altri componenti, e liberarla quando il componente ne ha terminato l'utilizzo.

Le API JDBC forniscono un'interfaccia standard chiamata `DataSource` per ottenere una connessione del pool. Ogni EJB container vendor deve fornire una implementazione di questa classe che si adatti al particolare tipo di pool utilizzato e faccia uso di uno o più algoritmi di caching per gestire l'assegnamento delle connessioni. Le istanze della classe che implementa questa interfaccia possono essere ottenute attraverso un nome logico utilizzando le solite API JNDI. In tal modo si garantisce nel codice anche un livello di astrazione rispetto al driver JDBC utilizzato e quindi rispetto al DBMS sottostante.
Nel codice dei componenti sarà quindi sufficiente effettuare un'operazione di lookup al nome logico JNDI corrispondente ad una particolare sorgente JDBC e ottenere così la classe che implementa l'interfaccia `DataSource`. Per ottenere la connessione vera e propria si utilizzerà il metodo `getConnection` dell'interfaccia che restituisce un normale oggetto `Connection` che riferisce ad una effettiva connessione aperta del pool e potrà essere utilizzato normalmente. Una volta terminato il suo utilizzo sarà sufficiente chiudere la connessione con il metodo `close` dell'oggetto `Connection` per restituirla al pool.
L'implementazione dell'interfaccia `DataSource` fornita dall'EJB container vendor utilizza altre interfacce JDBC, che devono anch'esse essere implementate dal vendor, di nome `PooledConnection` e `ConnectionPoolDataSource` e che sono gestite dal container in modo trasparente. In particolare quando il componente invoca il metodo `DataSource.getConnection`. l'EJB container effettua a sua volta un'operazione di lookup nel connection pool per vedere se esiste un'istanza di `PooledConnection` che può essere usata. Se esiste, questa viene ritornata al container e viene posto il lock. Se non esiste viene utilizzato un oggetto `ConnectionPoolDataSource` per ottenere una nuova istanza di `PooledConnection`. Quando il container ha ottenuto un'istanza di `PooledConnection` chiama il metodo `PooledConnection.getConnection` che restituisce un normale oggetto `Connection` che viene ritornato al componente.
Quando nel codice del componente viene chiamato il normale metodo `Connection.close` viene lanciato un evento che il container cattura e, invece di chiudere fisicamente la connessione, si limita a togliere il lock precedentemente posto.

```
// ottenere il context JNDI iniziale
Context initctx = new InitialContext ();

// ottenere l'implementazione del DataSource
DataSource ds = (DataSource)initctx.lookup("java :comp/env/jdbc/MyDatabase");

// ottenere una connessione dal pool
Connection con = ds.getConnection();

// utilizzo della connessione
// …

// restituzione della connessione al pool
```

con.close();

Frammento di codice per ottenere una connessione al pool

2.8 EJB-Centric e Web-Centric: confronto

La J2EE non pone alcun vincolo sul tipo di approccio da utilizzare per lo sviluppo di un'applicazione web multilivello.

Il classico approccio Web-centric, ancora molto diffuso per piccole applicazioni, prevede che sia la presentation logic sia la business logic risiedano sul Web tier e che questo si connetta direttamente al EIS tier tramite JDBC.

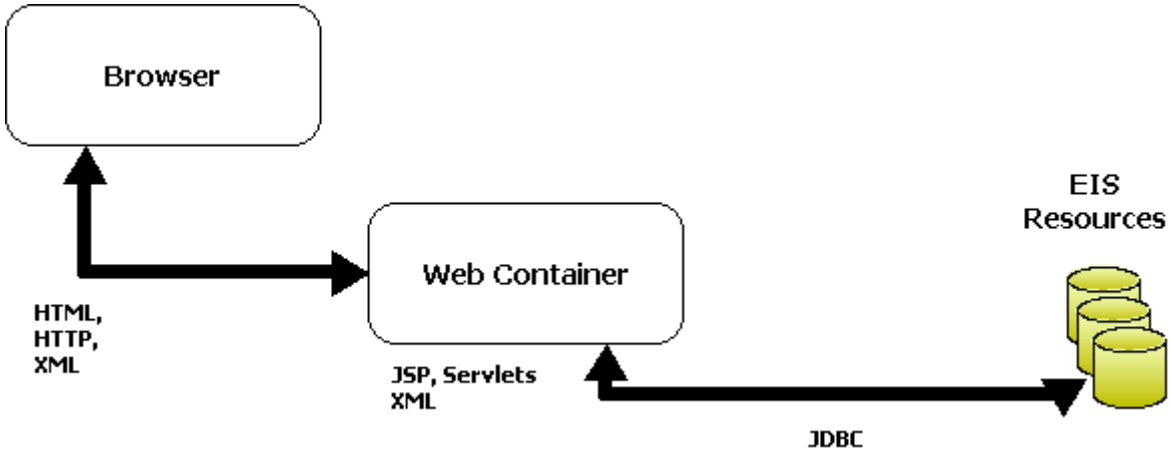


Figura 17-Approccio 3-tier web-centric

In questo approccio il Web-tier è responsabile di quasi tutta la funzionalità dell'applicazione. Infatti, oltre a processare le request del client per generare una opportuna response attraverso la produzione delle pagine web dinamiche, il Web-tier deve anche occuparsi di implementare tutte le funzionalità di aggiornamento dei dati nell'EIS-tier. Come conseguenza si ha che il Web-tier in questo tipo di approccio risulta essere notevolmente appesantito, tendendo a diminuire la scalabilità dell'applicazione.

Nell'approccio EJB-centric, come si è detto, tutta la business logic è lasciata all'EJB-tier ed è questo l'unico livello direttamente collegato all'EIS tier.

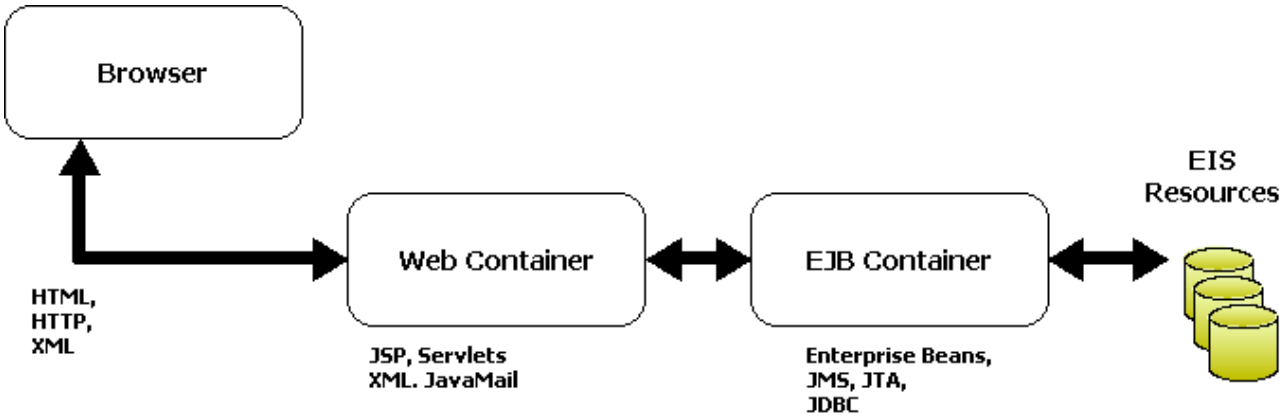


Figura 18-Approccio 3-tier EJB-centric

Il vantaggio principale di quest'ultimo approccio è costituito dalla possibilità di sfruttare le funzionalità messe a disposizione dall'EJB-container. Esso infatti, come si è detto, fornisce supporto per la gestione delle transazioni e per il connection pooling garantendo alti livelli di scalabilità, ideali per le applicazioni per le quali si preveda aumento di dimensioni. Inoltre questo approccio è particolarmente adatto per applicazioni distribuite grazie alla possibilità di suddividere Web tier e EJB tier su diversi hosts.

Conclusioni

La realizzazione di questo elaborato ha permesso di ampliare notevolmente il terzo capitolo della tesi "Un sito internet per la gestione dei progetti", ovvero quello riguardante l'ambiente di sviluppo. Nello sviluppo della tesi di cui sopra, infatti, mi ero soffermato maggiormente sulla parte riguardante la strutturazione delle informazioni da inserire nel database e sulle funzionalità che l'applicazione avrebbe dovuto fornire agli utenti.

MISP è stata realizzata basandosi su un'architettura a due livelli relativamente semplice rispetto all'architettura a tre livelli analizzata in questa trattazione, ed è emerso in modo chiaro come una progettazione che osservi le specifiche e sfrutti pienamente le funzionalità e gli strumenti della piattaforma Java 2 Enterprise Edition permetta di realizzare applicazioni notevolmente più strutturate (e quindi modificabili con maggiore semplicità), efficienti e potenti.

E' opportuno ricordare tuttavia che per progetti di piccole-medie dimensioni ed in mancanza di risorse adeguate, sia temporali che umane, lo sviluppo di applicazioni con un'architettura a due livelli non richiede particolari conoscenze e progettazioni pur garantendo un prodotto di qualità.

Concludendo, l'architettura 3-tier potrà ritenersi una valida opportunità da tenere in considerazione in funzione di eventuali sviluppi futuri di MISP già ipotizzati da Intertech Italia e riguardanti l'adozione dello strumento anche da parte dei clienti stessi della ditta.

Bibliografia

- Sito ufficiale di Java della Sun Microsystem: http://java.sun.com
- "Java 2 Enterprise Edition" documentazione disponibile al sito http://java.sun.com/j2ee/docs.html
- "Portale Web di Facoltà: progetto e implementazione di homepage dei docenti mediante architettura J2EE" di Andrea Cervellati disponibile al sito http://www.dbgroup.unimo.it
- "Un Sito Internet per la gestione dei progetti" di Filippo Battilani
- Schede tecniche Intertech Italia