Java EE

Una plataforma de components distribuïda

Josep Maria Camps Riba

PID_00185401

UOC
Universitat Oberta de Catalunya

Índex

Introducció

En els mòduls anteriors hem vist primer una introducció a la descripció arquitectònica dels sistemes distribuïts mitjançant punts de vista independents; després ens hem centrat a definir l'arquitectura de programari del sistema per desenvolupar i, finalment, hem vist com dissenyar els components que formaran el sistema sense tenir en compte la tecnologia concreta en què s'implementaran.

El darrer pas és implementar els components que formen el sistema distribuït que volem construir amb una tecnologia concreta (vista tecnològica). Ja hem vist una primera introducció a alguna plataforma distribuïda, com CORBA, i els fonaments de les comunicacions distribuïdes en Java en el mòdul "Java RMI" d'aquest material didàctic. En aquest mòdul descriurem amb força detall una de les tecnologies més emprades actualment per a implementar sistemes distribuïts empresarials: **Java Enterprise Edition (Java EE)**.

Analitzarem què és la plataforma Java EE, quins elements la formen i quin estil arquitectònic dels que hem vist en mòduls anteriors segueixen les aplicacions Java EE. Estudiarem la divisió per capes que proposa el model d'aplicacions de Java EE i veurem els components que ofereix la plataforma per a implementar les funcionalitats de cadascuna de les capes. Farem especial menció als **Enterprise Java Beans (EJB)**, els components de negoci distribuïts que ofereix Java EE.

També analitzarem les arquitectures més habituals en les quals es pot emmarcar una aplicació Java EE i quins components cal utilitzar en cada cas.

Finalment, mostrarem una manera de fer el disseny d'aplicacions Java EE amb UML, amb un exemple de perfil UML per a Java EE, i us donarem algunes recomanacions que heu de tenir en compte quan feu el pas de l'especificació a la implementació del sistema.

Objectius

L'objectiu bàsic del mòdul és mostrar a l'estudiant com s'implementarien els conceptes teòrics estudiats en els mòduls anteriors, concretant-los en una plataforma específica. Aquest objectiu més genèric es concreta en el següent:

1. Estudiar la plataforma Java EE com a exemple d'arquitectura o plataforma distribuïda de desenvolupament d'aplicacions empresarials.

2. Conèixer els conceptes de *component*, *contenidor* i *servei* (no confongueu el concepte de *servei* que es presenta en aquest mòdul amb el concepte de *servei* del mòdul "SOA"; en aquest mòdul ens referim a serveis que proporcionen els contenidors als components que tenen desplegats), tres conceptes clau per a entendre la filosofia inherent a la plataforma Java EE.

3. Veure quin és el model de programació d'aplicacions que proposa Java EE i establir la correspondència entre aquest model i els estils arquitectònics definits en el mòdul "Arquitectura del programari".

4. Analitzar la divisió per capes que proposa Java EE, quins components de la plataforma situem a cada capa i oferir, per a cada capa, un conjunt de recomanacions de disseny.

5. Fer una breu introducció al concepte de *framework* (marc de treball, en català) i veure'n exemples d'aplicació a cada una de les capes que proposa Java EE.

6. Estudiar els EJB com a components de negoci distribuïts, veure què són i què ofereixen, els diferents tipus que suporta Java EE i el cicle de vida de cadascun.

7. Estudiar diverses tecnologies que formen part de Java EE o hi estan relacionades.

8. Veure una possible manera de dissenyar, per a la implementació posterior, usant Java EE, una aplicació distribuïda (a partir de l'especificació independent de la tecnologia obtinguda tal com s'ha explicat en els mòduls anteriors).

1. Java EE, una plataforma de components distribuïda

Actualment, les empreses tenen un conjunt de necessitats força diferents de les que tenien fa uns anys. En l'era de la informació tota empresa que vulgui ser competitiva en el seu sector ha d'**oferir els seus serveis** als seus clients, empleats, socis, etc.

> Per exemple, avui dia totes les entitats financeres permeten als seus clients operar per Internet (fins i tot, hi ha entitats que **només** operen per Internet), oferint serveis que van des de l'obertura d'un compte d'estalvi fins a la consulta dels moviments associats a una targeta de crèdit.

Aquests serveis i els sistemes que els proporcionen han de satisfer una sèrie de requisits no funcionals:

- **Alta disponibilitat**. Els serveis que s'ofereixen no poden deixar de funcionar. S'han de proporcionar mecanismes per a assegurar que no deixin de funcionar i, si ho fan, que el temps d'interrupció sigui mínim.

- **Segurs**. S'ha de garantir que no hi hagi accessos no autoritzats als serveis i s'han d'establir polítiques d'accés per als diferents tipus d'usuaris d'aquests serveis.

- **Fiables**. Els serveis han de ser el màxim de fiables i lliures d'errors. S'ha d'oferir mecanismes de detecció i diagnosi d'errors.

- **Escalables**. Si augmenta la càrrega del sistema, ha de ser fàcil afegir servidors o bé ampliar els que ja tenim per donar la mateixa qualitat de servei sense haver de modificar les aplicacions existents.

- **Mantenibles**. S'han de poder afegir serveis als sistema i modificar els existents fàcilment.

- **Portables**. S'ha de poder canviar de plataforma d'una manera no traumàtica. Un canvi de plataforma no pot implicar tornar a implementar tots els serveis.

- **Ràpids de desenvolupar i de desplegar**. S'han de poder desenvolupar i oferir serveis als usuaris del sistema de manera àgil i ràpida. Avui dia, el famós *time-to-market* és vital per a les empreses.

- **Fàcilment integrables amb els sistemes existents**. Rarament es desenvoluparan serveis nous partint de zero. El normal serà integrar els nous ser-

veis amb serveis o desenvolupament ja existents, i els hem de poder integrar fàcilment.

Aquestes característiques són comunes a tots els serveis que volem oferir. La millor manera d'obtenir-les és utilitzant un conjunt de serveis de baix nivell que les proporcionin. Aquests es poden aconseguir de dues maneres:

1) fent servir una plataforma que els faciliti, o

2) carregant aquestes tasques de baix nivell i alta complexitat als desenvolupadors dels serveis de negoci.

Aquesta és la diferència bàsica entre utilitzar una plataforma de desenvolupament empresarial i no fer-ho. En el segon cas, els desenvolupadors perdran molt temps implementant serveis de baix nivell que tenen poc a veure amb els serveis de negoci que es volen oferir (i que són realment l'objectiu de l'empresa).

Per exemple, els desenvolupadors del servei de consulta de comptes d'una aplicació de banca electrònica no s'haurien de preocupar de desenvolupar el codi necessari perquè l'aplicació de consulta de comptes no permeti a un usuari consultar comptes que no siguin seus. Els desenvolupadors només haurien de definir les polítiques d'accés i deixar la implementació d'aquestes al servei de seguretat de la plataforma.

Per tant, necessitem una plataforma que ofereixi un conjunt de serveis als arquitectes i als desenvolupadors per tal de facilitar el desenvolupament d'aplicacions empresarials.

Java EE[1] és un exemple de plataforma de desenvolupament empresarial que té com a objectiu primordial fer més senzill i àgil el desenvolupament i la posada en marxa d'aplicacions empresarials. Per a aconseguir-ho Java EE ofereix un conjunt de llibreries i serveis que proporcionen a les aplicacions empresarials les característiques esmentades anteriorment (alta disponibilitat, seguretat, escalabilitat, etc.).

(1)De l'anglès *Java Enterprise Edition*.

En els apartats següents us donarem una visió d'alt nivell de Java EE. Us explicarem què és, quins elements en formen part, en què consisteixen, i quins són els conceptes clau per a poder entendre aquesta arquitectura. Farem especial menció a l'estil arquitectònic, al model de programació d'aplicacions i a alguns components de la plataforma, com poden ser els components web i els components de negoci.

1.1. Què és Java EE?

Java EE és una plataforma de desenvolupament empresarial (proposada per Sun Microsystems l'any 1997) que defineix un estàndard per al desenvolupament d'aplicacions empresarials multicapa.

Java EE **simplifica** el desenvolupament d'aquestes aplicacions basant-les en components modulars estandarditzats, proporciona un conjunt molt complet de serveis a aquests components i gestiona automàticament moltes de les funcionalitats o característiques complexes que requereix qualsevol aplicació empresarial (seguretat, transaccions, etc.), sense necessitat d'una programació complexa.

Es pot definir Java EE com:

> **Java EE** és una plataforma oberta i estàndard per a desenvolupar i desplegar aplicacions empresarials multicapa amb *n* nivells, distribuïdes i basades en components que s'executen en servidors d'aplicacions.

Així doncs, Java EE s'emmarca dins d'un **estil arquitectònic heterogeni**, aglutinant diverses característiques corresponents a estils arquitectònics en capes o nivells, client-servidor, orientada a objectes distribuïts i, fins i tot, orientada a serveis.

Vegeu també

Teniu la definició de tots aquests estils arquitectònics en el mòdul "Arquitectura del programari" d'aquesta assignatura.

D'aquesta definició podem extreure els conceptes bàsics i els punts clau que hi ha darrere la plataforma Java EE:

1) Java EE és una plataforma oberta i estàndard. No és un producte, sinó que defineix un conjunt d'estàndards que tots els contenidors han de complir per a comunicar-se amb els components. En alguns àmbits es diu que Java EE és una especificació d'especificacions.

Java EE

Una confusió que sovint es produeix és pensar que Java EE és un producte concret que distribueix una empresa i que es pot descarregar de la seva pàgina web (com podria ser el seu JDK). Això no és així. No tenim un Java EE concret, sinó diverses implementacions de les especificacions; no podeu anar a la web i descarregar "el Java EE".

Aquesta és una de les diferències primordials entre Java EE i la seva principal competidora en el mercat, **Microsoft .NET**: .NET defineix una plataforma propietària, mentre que Java EE defineix una plataforma oberta i estàndard. Aquest factor fa que hi hagi moltes implementacions de Java EE.

Microsoft .NET

Teniu una breu descripció de Microsoft .NET en el mòdul "Introducció a les plataformes distribuïdes" i podeu trobar molta més informació a la web de Microsoft.

Java EE és oberta i estàndard perquè la plataforma es desenvolupa i s'actualitza mitjançant el JCP[2], que és el procés formalitzat responsable de **totes** les tecnologies de la plataforma Java. Totes les tecnologies que formen part de Java EE es "creen" mitjançant JSR[3] validats pel JCP. Grups d'experts de tots els àmbits de la comunitat treballen en la creació i validació dels JSR que després ratificarà el JCP i passaran a formar part de la plataforma Java.

(2)De l'anglès *Java Community Process*.

(3)De l'anglès *Java Specification Request*.

JCP

El JCP (*Java Community Process*) és un organisme format per prop de 1.200 empreses, associacions i particulars, amb l'objectiu de vetllar per una evolució correcta i coherent de les plataformes basades en Java.

Un JSR és una petició d'especificació d'una tecnologia Java que es fa al JCP. Si el comitè l'aprova es crea el document d'especificació, el test de compatibilitat i la implementació de referència.

2) Java EE defineix un model d'aplicacions distribuït i multicapa amb *n* nivells. Amb aquest model podem dividir les aplicacions en parts i cadascuna d'aquestes parts es pot executar en servidors diferents. L'arquitectura Java EE defineix un mínim de 3 capes: la capa client, la capa de servidor i la capa de sistemes d'informació de l'empresa (EIS[4]). La capa EIS també s'anomena *capa de dades*.

(4)De l'anglès *Enterprise Information Systems*.

3) Java EE basa les aplicacions empresarials en el concepte de components modulars i estanderditzats. Aquest concepte està molt lligat al concepte *contenidor*. Els contenidors són entorns estàndard d'execució que proporcionen un conjunt de **serveis** als components que estalvien molta feina a l'hora de desenvolupar els components.

Nota

No us preocupeu si no enteneu conceptes com *component, contenidor, servei, desplegament, descriptor de desplegament*, etc. Al llarg del mòdul els veurem a fons.

Per exemple, el servei de transaccions ens permetrà definir un component amb comportament transaccional sense haver de codificar ni una línia de codi transaccional. El desenvolupador indicarà el comportament transaccional del component en un descriptor XML, o amb anotacions, i el contenidor on es desplegui el component s'encarregarà de cridar el servei de transaccions de la plataforma per a implementar aquest comportament.

1.1.1. Java EE dins la plataforma Java

Java EE és el resultat d'un esforç fet per Sun Microsystems per alinear les diferents tecnologies i API Java existents en una plataforma unificada de desenvolupament d'aplicacions empresarials.

Hi ha quatre plataformes per al llenguatge de programació Java:

1) *Java Platform, Standard Edition* (Java SE). Aquesta és la plataforma de desenvolupament de Java que tothom coneix, i que permet executar desenvolupaments fets en Java a qualsevol ordinador que tingui instal·lada una màquina virtual Java (JVM). Aquesta plataforma conté les funcionalitats principals.

2) *Java Platform, Enterprise Edition* (Java EE). Plataforma Java per a desenvolupar i desplegar aplicacions empresarials que requereixen un conjunt de característiques complexes com seguretat, transaccionalitat, robustesa, alta disponibilitat, etc. Aquesta plataforma s'ha definit prenent com a base la plataforma Java SE.

3) *Java Platform, Micro Edition* (Java ME). Plataforma de desenvolupament per a dispositius mòbils (PDA, telèfons mòbils, *paggers*, etc.), amb capacitat per a executar Java. Es tracta d'una versió reduïda de la plataforma estàndard que té en compte les limitacions de memòria, capacitat i rendiment d'aquests dispositius.

4) Java FX. Plataforma per a crear aplicacions RIA[5] escrites en Java FX Script.

(5)De l'anglès *Rich Internet Applications*.

Totes les plataformes Java esmentades estan formades per una màquina virtual de Java i un conjunt d'API, i això permet que les aplicacions escrites per a aquesta plataforma es puguin executar en qualsevol sistema compatible amb la plataforma, gaudint de tots els avantatges que ofereix el llenguatge de programació Java.

1.1.2. Java EE i Java EE

A partir de la versió 5 de la plataforma Java s'han simplificat els noms. El "2" s'elimina del nom i també s'elimina el número i el punt de la versió. Amb això la versió 6 de la plataforma Java per a desenvolupar aplicacions empresarials és *Java Platform, Enterprise Edition 6* (Java EE 6).

Les versions 5 i 6 de la plataforma també han introduït un conjunt de canvis importants respecte a les versions anteriors amb l'objectiu de simplificar-ne el model de desenvolupament. Entre els canvis més rellevants podem destacar els següents:

- Ús extensiu de les noves capacitats introduïdes en la versió de Java SE.
- Els descriptors de desplegament passen a ser opcionals i es pot posar aquesta informació amb anotacions en el codi font.
- Ús de la injecció de dependències per a simplificar el desenvolupament, reduir l'acoblament i facilitar el test unitari i el desplegament.
- Ús de JPA per a modelitzar la persistència d'entitats.
- Ús de perfils per no haver de suportar totes les tecnologies Java.
- Simplificacions en el model d'assemblatge i desplegament de les aplicacions.

JPA

JPA és l'API de persistència estàndard que defineix Java. En el subapartat "Entitats JPA" d'aquest mòdul s'explica en detall aquesta tecnologia.

1.2. Model de programació d'aplicacions Java EE

La plataforma Java EE fa servir un model d'aplicació distribuïda, basada en components i multicapa amb *n* nivells.

El model de programació d'aplicacions Java EE defineix un conjunt de recomanacions de disseny per a dividir una aplicació Java EE en capes i decidir quins components cal posar a cada capa. El model de programació defineix l'estructura que han de seguir les aplicacions que es desenvolupen sota la plataforma i tot un conjunt de bones pràctiques a l'hora de fer servir la plataforma.

Java EE proposa una arquitectura multicapa com a estil arquitectònic per a les aplicacions que es desenvolupin sota aquesta plataforma. Això vol dir que per a crear una aplicació típica Java EE cal dividir-la en capes, desenvolupar els components que calguin i col·locar-los a la capa corresponent.

Per a entendre quin és el model de programació d'aplicacions de Java EE hem d'aprofundir en els conceptes de *component*, *contenidor* i *servei* que ja heu vist en els materials, i establir la relació que hi ha entre aquests i el model de programació d'aplicacions Java EE.

1.2.1. Components, contenidors, serveis i servidors d'aplicacions

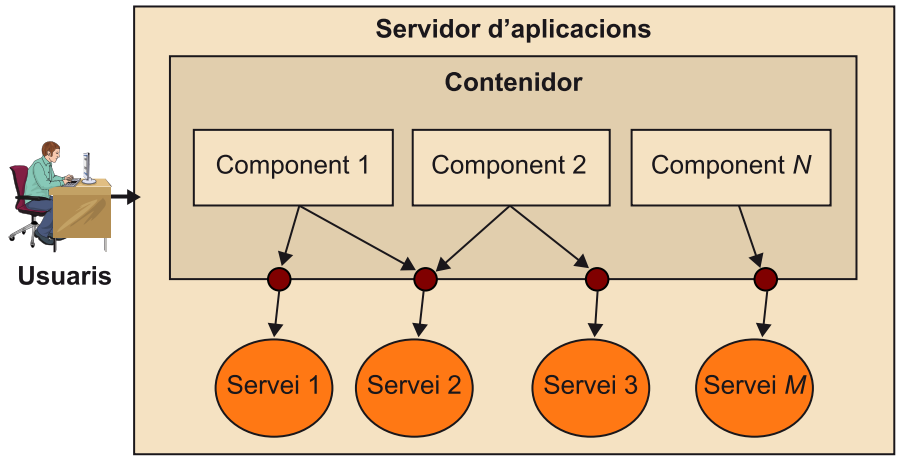

Figura 1. Components, contenidors, serveis i servidors d'aplicacions

Una aplicació Java EE estarà formada per un conjunt de **components** que s'assemblen i es despleguen en contenidors Java EE, per tal d'oferir als usuaris la funcionalitat desitjada.

Els **contenidors** són peces de programari que ofereixen un conjunt de serveis als components que hi estan desplegats.

Els **servidors d'aplicacions** són peces de programari que implementen els serveis que ofereixen els contenidors als components. Els contenidors formen part dels servidors d'aplicacions.

La majoria d'aplicacions empresarials necessita un conjunt de serveis de baix nivell idèntics: gestió del cicle de vida dels components, seguretat, transaccions, balanceig de càrrega, gestió de recursos, etc.

Una primera opció és que cada aplicació implementi aquests serveis, però, com que aquests serveis es repeteixen en totes les aplicacions que desenvolupem, és convenient definir-los i implementar-los una sola vegada i deixar que totes les aplicacions en puguin fer ús.

Les aplicacions Java EE es construeixen assemblant components i desplegant-los en un contenidor. Les aplicacions estan formades per components que s'executen dins de contenidors. Els contenidors proporcionen un entorn d'execució i accés a un conjunt de serveis de baix nivell als components que formen l'aplicació.

El procés d'**assemblatge** d'un component consisteix a empaquetar en un format estàndard tot el que forma el component (classes, interfícies, descriptors, etc.) per a poder fer-ne el desplegament.

El **desplegament** d'un component és la fase d'instal·lació del component al contenidor corresponent. En aquesta fase es configuren els serveis del contenidor que necessita el component. La configuració dels serveis usats pels components es pot fer declarativament amb fitxers XML que s'empaqueten amb els components o, darrerament, amb anotacions.

Per exemple, un component pot configurar la seguretat de manera declarativa en ser desplegat en el contenidor per a especificar quins usuaris o grups d'usuaris estan autoritzats a executar quins mètodes del component.

En un model de components i contenidors, els components tenen la lògica de negoci i els contenidors els proporcionen un entorn d'execució i tot un conjunt de serveis de baix nivell.

És important tenir en compte que en un model de components i contenidors els clients interactuen amb els contenidors en lloc de fer-ho amb els components directament. Els contenidors intercepten les crides als components, executen les tasques que s'han definit declarativament per al component i li passen el control perquè executi la lògica programada per aquesta crida.

Per tal que els contenidors puguin gestionar el cicle de vida, proporcionar un entorn d'execució i oferir un conjunt de serveis als components, s'han de definir una sèrie de normes (un contracte estàndard) que els components i els contenidors han de complir.

> Per exemple, els *servlets*, un tipus de component de la plataforma Java EE que es desplega dins un contenidor web, defineixen un contracte amb el contenidor web segons el qual totes les peticions d'usuari cap al component executaran un mètode anomenat *service()*. Veureu els *servlets* de servidor i el contenidor web en detall en el subapartat "Capa de presentació" d'aquest mòdul.

Java EE defineix un model de components i contenidors **obert** i **estàndard**. Això vol dir que els contractes entre els components, els contenidors i els serveis que han de proporcionar els defineix una **especificació estàndard**.

.NET

.NET també segueix un model de components i contenidors però, a diferència de Java EE, els contractes entre components i contenidors són propietaris. Per això, només Microsoft pot desenvolupar contenidors i servidors compatibles amb .NET.

Així s'aconsegueix que qualsevol fabricant pugui desenvolupar contenidors capaços d'executar components Java EE; només ha de complir els contractes estàndard i implementar els serveis requerits per l'especificació. Les plataformes de programari que implementen aquests contenidors i serveis s'anomenen ***servidors d'aplicacions*** i n'hi ha de molts fabricants diferents, alguns de comercials, altres de codi lliure.

La manera que té Oracle de certificar que una implementació és compatible amb l'especificació de Java EE és fent que el fabricant del servidor d'aplicacions passi un conjunt de tests anomenats *test de compatibilitat*.

La figura 2 mostra alguns dels tipus de components i contenidors que podem trobar en la plataforma Java EE.

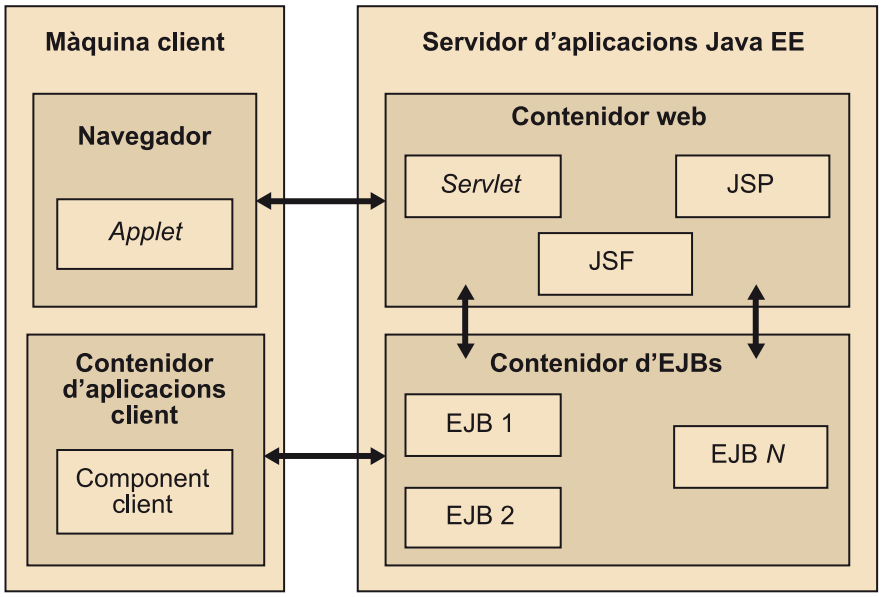

Figura 2. Components i contenidors

Components

En el context particular de Java EE es defineixen els components següents:

- **Components client**. Un component client és un programa que s'executa en un entorn client, ja sigui com a miniaplicació o bé com a aplicació Java autònoma.

Java autònom

Una aplicació Java autònoma és una aplicació Java independent, i pot ser tant una aplicació de consola com una aplicació amb interfície gràfica.

- **Components web**. Un component web és una peça de programari que genera una resposta a una petició HTTP. Java EE defineix dos tipus de components web, els ***servlets*** i les **pàgines web** (***Java Server Faces*** o ***Java Server Pages***).

- **Components de negoci**. Un component de negoci és una peça de programari que implementa algun concepte de la lògica del domini particular de negoci en el qual s'està desenvolupant l'aplicació. Java EE defineix els **EJB**[6] com a components de negoci, i poden ser de dos tipus: **EJB de sessió** i **EJB de missatge**. A part dels EJB, els components de negoci també es poden implementar amb classes Java.

[6] De l'anglès *Enterprise Java Beans*.

Contenidors

En el context particular de Java EE es defineixen els contenidors següents:

- **Contenidor d'aplicacions**. Conté aplicacions Java autònomes. El trobem en les màquines client amb aplicacions Java.

- **Contenidor d'*applets* (miniaplicacions)**. Proporciona un entorn d'execució per a miniaplicacions. El trobem als navegadors de les màquines client.

- **Contenidor web**. Proporciona un entorn d'execució per als components web que defineix la plataforma Java EE, *servlets* i pàgines web. El trobem en el servidor.

- **Contenidor d'EJB**. Proporciona un entorn d'execució per als EJB, els components de negoci que ofereix Java EE. El trobem en el servidor.

Serveis

Els contenidors proporcionen als components un conjunt de serveis que poden ser configurables o no configurables.

- Els serveis **configurables** permeten als desenvolupadors especificar el comportament dels components en temps de desplegament de l'aplicació. Entre aquests podem citar els serveis de transaccions, de seguretat i el servei de noms.

- Els contenidors també ofereixen serveis **no configurables**, com la gestió del cicle de vida dels components, la gestió de les fonts de connexions a recursos externs, l'accés a les API de Java EE, etc.

API

Application Program Interface (API) són les interfícies que proporciona un sistema de programari (component, servei, aplicació, etc.) perquè els clients hi puguin interactuar. Per exemple, l'API de JDBC (*Java Database Connectivity* és una API per a accedir a bases de dades relacionals) ens ofereix un conjunt de mètodes que els clients poden utilitzar per a interactuar amb una base de dades relacional (obrir una connexió, fer una consulta, fer una actualització, etc.).

Els contenidors poden accedir als serveis mitjançant les API estàndard de cada servei que defineix l'especificació de Java EE. Per exemple, l'accés al servei de transaccions es fa mitjançant l'API *Java Transaction API* (JTA).

Vegeu també

En el subapartat "Plataforma Java EE" d'aquest mòdul teniu una explicació més extensa dels serveis, API i tecnologies que proporciona l'especificació de Java EE.

Servidors d'aplicacions

Els **servidors d'aplicacions** Java EE són peces de programari que implementen els contenidors, serveis i API que defineix la plataforma Java EE. Fan de "cola" entre tots aquests elements, proporcionant un entorn de gestió, desplegament i execució integrat per a les aplicacions empresarials desenvolupades sota la plataforma Java EE.

Els servidors d'aplicacions fan que els components, contenidors i serveis estiguin sincronitzats i puguin treballar conjuntament per a proporcionar una sèrie de funcionalitats. En els servidors d'aplicacions residiran els components empresarials, bé siguin objectes distribuïts accessibles remotament, components web, pàgines web o, fins i tot, miniaplicacions (*applets*).

És important remarcar que qualsevol aplicació Java EE s'ha de desplegar en un servidor d'aplicacions perquè els clients hi puguin interactuar.

Qualsevol servidor d'aplicacions que implementi les especificacions de la plataforma Java EE ha d'oferir **totes** les tecnologies, API i serveis que estiguin inclosos com a obligatoris en l'especificació. Ja hem vist que aquesta premissa és clau, ja que proporciona al desenvolupador un conjunt de serveis estàndard disponibles i li permet centrar-se en la lògica de negoci, sense preocupar-se d'implementar els serveis de baix nivell.

Noves tecnologies en Java EE

Amb el pas del temps la plataforma Java EE s'ha anat engreixant per a incorporar noves tecnologies a cada versió. En la versió 6 ha introduït el concepte de *perfils* per tal de restringir el nombre de tecnologies necessàries en una instància particular de la plataforma i així no haver de suportar tecnologies que no seran de cap utilitat per a aquesta instància particular. Per exemple, s'ha definit el perfil web (Java EE *Web Profile*) amb les tecnologies que han d'implementar les instàncies que únicament vulguin suportar aplicacions web.

Un servidor d'aplicacions Java EE tindrà, normalment, un contenidor web, un contenidor d'EJB, un sistema de missatgeria i moltes eines que ens ajudaran a augmentar la productivitat en el desenvolupament d'aplicacions.

El fet que Java EE es basi en especificacions públiques fa, tal com hem vist, que hi hagi múltiples implementacions de servidors d'aplicacions. Actualment, el mercat dels servidors d'aplicacions és un dels mercats de productes de programari més actius i hi ha moltes empreses que llencen els seus productes i lluiten per oferir cada cop més prestacions. També hi ha diverses implementacions de servidors d'aplicacions de codi lliure, que no tenen res a envejar als seus homònims comercials.

Com a exemples de servidors d'aplicacions podem esmentar:

- JBoss
- Jetty
- ApacheGeronimo
- IBM WebSphere
- Oracle Application Server
- Oracle WebLogic
- GlassFish

Servidors d'aplicacions

Hi ha una llista completa de servidors d'aplicacions que han passat els tests de compatibilitat per a les diferents versions de la plataforma Java EE a la web d'Oracle.

Una pregunta que us podríeu fer és: quina diferència hi ha entre dos servidors d'aplicacions si tots han d'implementar el mateix conjunt d'especificacions? Doncs, a part que cada servidor pugui oferir més o menys serveis de valor afegit, cada servidor implementa les especificacions a la seva manera i això fa que alguns siguin més ràpids, altres més robustos, etc.

Fins ara hem vist una breu introducció a Java EE i, dins el model de programació d'aplicacions, hem analitzat els conceptes de component, contenidor, servei i servidor d'aplicacions. Ara passarem a veure sota quin estil arquitectònic podem emmarcar les aplicacions que desenvolupem amb la plataforma Java EE.

1.2.2. Arquitectura multicapa i multinivell

En aquest apartat veurem què vol dir que una aplicació Java EE és una aplicació multicapa amb *n* nivells, i diferenciarem capes i nivells.

Cal que tingueu en compte que el disseny d'una aplicació Java EE seguint una arquitectura multicapa, tal com especificarem en aquests materials, no és adient per a tots els tipus d'aplicació Java EE. Aquesta arquitectura només és adient per a aplicacions empresarials complexes, amb molta lògica de negoci i amb un temps de vida força llarg. Per a aplicacions petites, molt orientades a dades i amb poca càrrega de negoci potser que aquesta arquitectura no sigui la més adient per la complexitat que afegeix al desenvolupament; en aquests casos potser Java EE tampoc és l'opció més adient. Hi ha tecnologies RAD més adients per a desenvolupar aplicacions que tinguin aquests requisits.

RAD

RAD és l'acrònim anglès de *Rapid Application Development*, i és un mètode de desenvolupament d'aplicacions que es basa en el desenvolupament interactiu, la construcció de prototipus i l'ús d'eines CASE (*Computer Aided Software Engineering*).

El model d'aplicacions que defineix Java EE s'emmarca dins un estil arquitectònic heterogeni que combina característiques de diferents estils, centrant-se en un estil client-servidor, basat en components i organitzat en capes.

Vegeu també

Podeu consultar aquests estils arquitectònics en el mòdul "Arquitectura del programari" d'aquesta assignatura.

És molt important distingir els conceptes de capes (*layers*) i nivells (*tiers*); moltes vegades es confonen aquests dos termes i això pot donar lloc a confusions a l'hora de definir l'arquitectura de les aplicacions. Les capes s'ocupen de la divisió **lògica** dels components que formen part de l'aplicació i no tenen en compte la localització física d'aquests components. Els nivells s'ocupen de la distribució **física** dels components que formen part de l'aplicació en diferents màquines. Tant les capes com els nivells utilitzen noms similars (*presentació*, *negoci*, *dades*, etc.) i això pot portar a confusions.

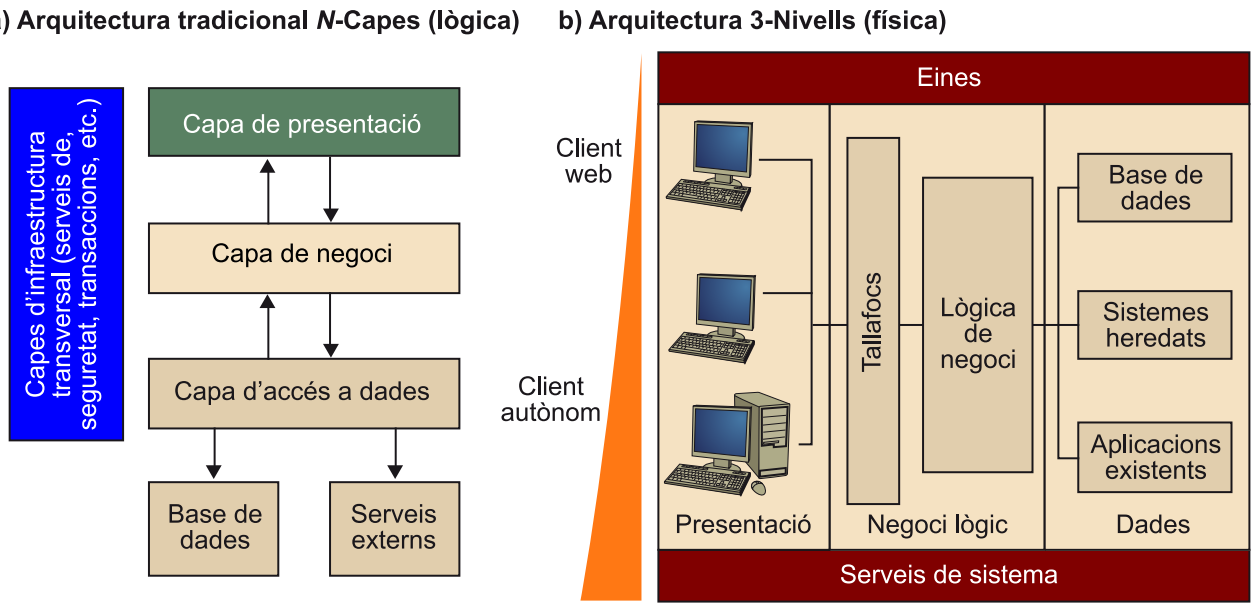

Figura 3. Arquitectura lògica i arquitectura física

De manera general, es pot afirmar que qualsevol aplicació complexa hauria de seguir una arquitectura lògica multicapa, mentre que no totes les aplicacions complexes requereixen una arquitectura física amb més d'un nivell.

1.2.3. Arquitectura lògica

En aquest apartat veurem l'arquitectura lògica de les aplicacions Java EE. Veurem què vol dir que una aplicació Java EE és una aplicació multicapa, analitzarem amb detall quantes capes defineix el model d'aplicacions Java EE, quines són, quins contenidors hi ha en cada capa i quins components podem desplegar en cadascun d'aquests contenidors.

La bona distribució d'una aplicació en capes incrementa la mantenibilitat, la robustesa, l'extensibilitat, l'escalabilitat i la seguretat de l'aplicació. La distribució en capes d'una aplicació depèn de molts factors i la divisió que es presenta en aquests materials només és una proposta que us pot servir de base per a cada cas concret.

Heu de veure les capes com a agrupacions lògiques de components de programari que formen part de l'aplicació. Un concepte clau en la distribució per capes d'una aplicació és que els components d'una capa només poden interactuar amb components transversals, components de la mateixa capa o amb components de les capes inferiors (ja sigui només amb la capa immediatament inferior, si seguim un enfocament estricte, o amb qualsevol capa inferior, en un enfocament menys restrictiu).

En general, podem considerar que una aplicació JEE es pot dividir en les capes que mostra la figura 4, cadascuna amb unes responsabilitats diferenciades.

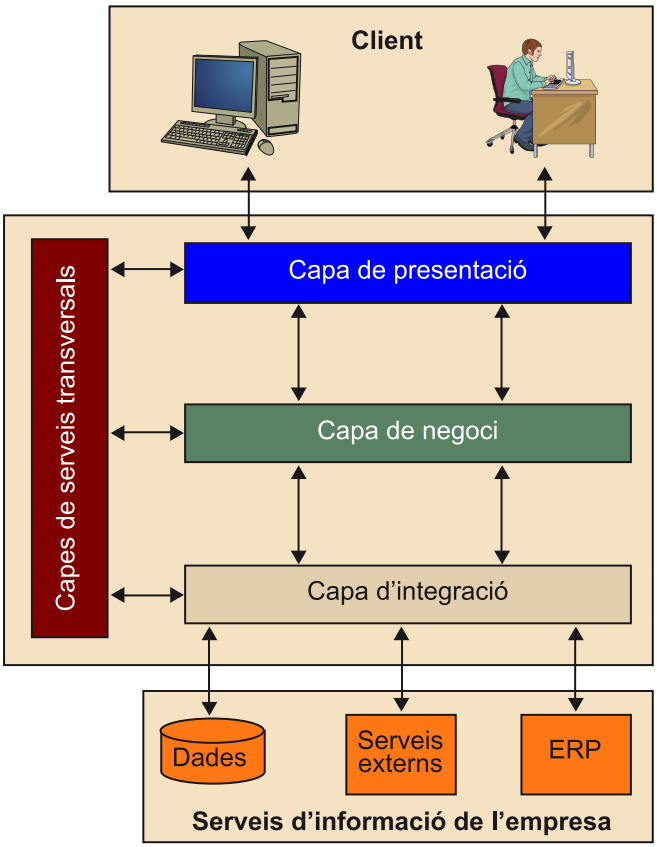

Figura 4. Arquitectura lògica d'un sistema multicapa

Els noms de les capes

Els noms de les capes poden canviar depenent de la publicació que es consulti. Aquests noms són els que hi ha al llibre:

D. Alur; **J. Crupi**; **D. Malks** (2003). *Core J2Ee Patterns: Best Practices and Design Strategies* (2a. Ed.). Upper Saddle River: Prentice Hall.

Usuari

Ens referim a **usuari** en el sentit ampli de la paraula. Tant poden ser usuaris humans com altres sistemes de programari.

En aquest esquema veiem que tenim quatre capes lògiques, els clients i els sistemes d'informació de l'empresa. Vegem primer què són els clients i els sistemes d'informació de l'empresa i després analitzarem amb detall les 4 capes lògiques (presentació, negoci, integració i serveis transversals) en què dividirem les aplicacions.

Clients

Java EE suporta molts tipus diferents de clients, que es comunicaran amb l'aplicació per a proporcionar als usuaris la funcionalitat de negoci desitjada.

Aquests clients es poden executar en diversos dispositius, com poden ser telèfons mòbils, ordinadors personals, PDA, etc. Ara veurem quins tipus de clients suporta la plataforma Java EE i com es comunica cadascun amb l'aplicació.

Una aplicació Java EE pot tenir tres tipus de clients:

1) Les miniaplicacions (*applets*).

2) Els clients web *(thin client*[7]*)*.

3) Els clients d'escriptori, ja siguin clients Java o clients no Java.

(7)En aquests materials quan parlem de *thin client* ens estem referint als navegadors web (Internet Explorer, Mozilla, Opera, etc.).

Les miniaplicacions són programes fets en Java que s'executen a la màquina virtual que tenen els navegadors i que es descarreguen dels servidors per a ser executats. Normalment es descarreguen per HTTP i s'executen incrustats en una pàgina web.

Miniaplicacions

Les miniaplicacions no solament es poden executar incrustades al navegador, sinó que ho poden fer en totes les aplicacions o dispositius que suportin el model de programació de les miniaplicacions.

La principal utilitat de les miniaplicacions és proporcionar una riquesa visual i operativa a la interfície d'usuari que, fins fa poc, seria molt difícil d'aconseguir amb una interfície amb HTML.

Els clients web són navegadors que consumeixen pàgines HTML o XML enviades pel servidor al navegador i que aquest s'encarrega de renderitzar. Les pàgines HTML o XML que formen la interfície d'usuari per a un client web les genera dinàmicament un component web de la capa de presentació i el servidor d'aplicacions envia el resultat d'aquesta generació al navegador que li ha fet la petició. La comunicació entre el client i el servidor es fa, normalment, en una o diverses peticions HTTP.

El **client web** és el tipus de client per excel·lència en les aplicacions Java EE.

Els clients Java s'executen en el seu contenidor propi, en una màquina virtual instal·lada al client. Les aplicacions client s'empaqueten amb un format determinat i s'han d'instal·lar explícitament a la màquina client. Els clients Java accedeixen directament als components de la capa de negoci del servidor.

Aplicacions Java

Les aplicacions Java també es poden instal·lar mitjançant la tecnologia Java Web Start.

La manera més usual de comunicar un client no Java amb un component de servidor desenvolupat amb Java EE és fent servir **serveis web**.

Vegeu també

Veureu una introducció als serveis web en el mòdul "SOA" d'aquest material didàctic.

És molt important fer una bona elecció del tipus de client que es proporciona amb l'aplicació, ja que és la "cara" de l'aplicació, la part amb la qual interactuarà l'usuari. Cal aconseguir que el client escollit reuneixi el conjunt de característiques idònies per al tipus d'aplicació que es desenvolupa.

Sistemes d'informació de l'empresa

Els **sistemes d'informació de l'empresa (EIS)** són tots els recursos als quals s'accedirà des de la capa d'integració.

Aquests sistemes són totalment heterogenis i gairebé sempre estaran separats físicament de l'aplicació (poden, fins i tot, ser serveis externs). Poden ser des de servidors de bases de dades relacionals, bases de dades orientades a objectes, ERP, transaccionals, sistemes SAP[8], sistemes de fitxers, serveis externs, etc.

(8)**SAP** és una empresa que comercialitza un ERP.

ERP

ERP vol dir *Enterprise Resource Planning*, una arquitectura de programari empresarial que integra la informació i els processos de manufactura, logística, finances i recursos humans d'una empresa.

Capa de presentació

La capa de presentació pot tenir components a la banda del client per a les aplicacions d'escriptori i les miniaplicacions; o a la banda del servidor, dins un contenidor web, per a les aplicacions web. Independentment de si la tenim al client o al servidor, la capa de presentació s'encarregarà de controlar la generació del contingut que s'enviarà a l'usuari perquè aquest interactuï amb l'aplicació. La capa de presentació es comunica amb lògica de negoci per a aconseguir les dades i assemblar-les, i proporcionar així les vistes adequades a cada perfil d'usuari.

En aquests materials ens centrarem en la capa de presentació per a aplicacions web.

La **capa de presentació** d'una aplicació web proporciona accés a la capa de negoci de l'aplicació als clients que accedeixen per HTTP i genera les vistes amb les quals interactuaran els clients. Com que ens centrem en la capa de presentació de les aplicacions web, podem anomenar la capa de presentació *capa web*.

Capa web

No és obligatori que la capa web cridi la capa de negoci. En aplicacions simples pot passar que el mateix component web tingui codificat la lògica de negoci.

En una arquitectura per capes, la capa web s'encarrega de rebre les peticions HTTP dels clients, invocar la lògica de negoci adient i generar la resposta que s'envia al client. La capa web pot generar qualsevol tipus de contingut, encara que normalment és HTML o XML.

Per a proporcionar les funcionalitats anteriors cal desenvolupar, empaquetar i desplegar el que s'anomena ***aplicació web***.

Una aplicació web està formada per un conjunt de components, contingut estàtic i fitxers de configuració que s'empaqueten (assemblen) i despleguen com una unitat. Les aplicacions web s'empaqueten en fitxers *.war* amb una estructura estàndard i es despleguen en un contenidor web de la plataforma Java EE perquè els clients hi tinguin accés.

.war

Un fitxer *.war* (*Web Archive*) segueix una estructura estàndard i conté els components, el contingut estàtic i els fitxers de configuració de l'aplicació web. Aquest fitxer és el que es desplega en el contenidor web.

El contenidor web és l'encarregat de gestionar el cicle de vida dels components que formen l'aplicació web, passar les peticions dels clients als components i donar accés al context d'execució.

L'especificació de la plataforma Java EE defineix el contracte entre el contenidor web i el component web, i defineix el cicle de vida dels components web, les possibles interaccions entre els components i el contenidor web, i els serveis que ha d'oferir el contenidor al component.

Aquest factor i el fet que la manera d'empaquetar les aplicacions web estigui estandarditzada fa que els components web i les aplicacions web que els contenen siguin portables entre diferents implementacions de contenidor web i es puguin desplegar en qualsevol servidor d'aplicacions sense haver de recompilar el component ni regenerar el fitxer que empaqueta l'aplicació web.

Portabilitat

La portabilitat ens permet, per exemple, agafar una aplicació web empaquetada en un fitxer *.war* desplegat en un servidor d'aplicacions **Jboss** i desplegar-la en un contenidor web **Tomcat** sense fer cap modificació.

La plataforma Java EE defineix dos tipus de components web:

- ***Servlets***. Classe Java que rep peticions HTTP, les processa i genera respostes a aquestes peticions.

- **Pàgines web**. Documents de text creats utilitzant tecnologia Java: JSF[9] o JSP[10], que s'executen com a *servlets* però permeten una aproximació més natural a la creació de contingut estàtic.

(9)De l'anglès *Java Server Faces*.

(10)De l'anglès *Java Server Pages*.

JSF

JSF és un tecnologia que pren com a base els *servlets* i proporciona un *framework* de components d'interfície d'usuari per les aplicacions web.

Les pàgines HTML estàtiques, els *applets* i les classes d'utilitat s'empaqueten amb els components web, però no són considerats com a tals per l'especificació Java EE.

Una opció comuna per a implementar la capa web d'una aplicació Java EE és utilitzar un *framework*[11] web que implementi un patró **Model-Vista-Controlador**.

(11)Els *frameworks* s'anomenen *marcs de treball* en català.

Vegeu també

En assignatures anteriors de l'àrea d'enginyeria del programari teniu una descripció d'aquest patró de disseny.

Capa de negoci

La capa de negoci és, segurament, la capa més important dins d'una aplicació empresarial. Aquesta capa contindrà els components, les relacions i les regles que resolen els processos de negoci de l'empresa.

Subcapes de negoci

Depenent de l'enfocament seguit en el disseny de l'aplicació, la capa de negoci es pot dividir en subcapes amb diferents responsabilitats. Per exemple, en un disseny guiat pel model del domini podem dividir la capa de negoci en capa d'aplicació i capa de domini.

La **capa de negoci** proporciona als seus clients les funcionalitats de negoci específiques en un domini particular. En la plataforma Java EE la lògica de negoci s'implementa amb components de negoci reutilitzables anomenats *EJB*[12], que s'empaqueten, es despleguen i s'executen en el contenidor d'EJB del servidor d'aplicacions.

(12)De l'anglès *Enterprise Java Beans*.

Els EJB són una de les tecnologies més importants de l'especificació de Java EE. Es tracta d'una tecnologia de components de servidor que permet construir aplicacions empresarials basades en Java. El desenvolupament de la lògica de negoci amb EJB fa que les aplicacions siguin escalables, transaccionals, portables, distribuïdes i segures.

Vegeu també

En el subapartat "Capa de negoci" d'aquest mòdul analitzarem amb detall la tecnologia d'EJB amb els diferents tipus i la utilitat de cadascun.

Tal com hem vist amb les aplicacions web, els EJB s'empaqueten en fitxers amb una estructura estàndard i s'han de desplegar dins un contenidor d'EJB de la plataforma Java EE perquè els clients hi tinguin accés.

L'especificació d'EJB 3.x defineix dos tipus bàsics d'EJB per a la capa de negoci:

1) EJB de sessió
2) EJB de missatge

Es recomana utilitzar els EJB de sessió per a implementar la capa de negoci, encara que cada cop més s'accepten altres alternatives com els **POJO**[13].

(13)De l'anglès *Plain Old Java Objects*.

POJO

Fins a la versió 1.4 de Java EE, Sun només acceptava l'ús de components EJB de sessió per a implementar la lògica de negoci i d'EJB d'entitat per a modelitzar les dades. A partir de l'especificació 1.4 de Java EE ja s'accepten altres tipus d'implementació per a la lògica de negoci com, per exemple, POJO.

La gran avantatja dels POJO és que són objectes molt lleugers que no afegeixen cap tipus de càrrega addicional (gestió de transaccions, seguretat, cicle de vida, etc.). Això els fa molt fàcils de desenvolupar, a més de proporcionar molt bon rendiment. El seu principal inconvenient és que no tenim cap de les característiques que tenen els EJB i si les necessitem les haurem d'implementar (o fer servir algun *framework* que en proporcioni).

POJO i POJI

Els POJO (*Plain Old Java Object*) són classes Java normals i els POJI (*Plain Old Java Interface*) són interfícies Java normals.

Capa d'integració

La **capa d'integració** s'encarrega d'accedir a les dades, que poden estar emmagatzemades en fonts molt heterogènies (bases de dades relacionals, sistemes llegats, bases de dades orientades a objectes, transaccionals, ERP, etc.).

La capa d'integració obté les dades de les diferents fonts de dades i les passa a la capa de negoci perquè aquesta darrera faci les transformacions adients en cada cas.

Normalment la divisió en capa de negoci i capa d'integració sol ser purament lògica, és a dir, els components de les dues capes són físicament a la mateixa màquina i, possiblement, desplegats en el mateix contenidor.

En la plataforma Java EE hi ha diverses aproximacions possibles per a representar les entitats (dades) sempre que parlem d'accés a bases de dades relacionals:

- **POJO**. Fer servir classes Java normals que utilitzin alguna tècnica per a accedir a les dades (JDBC directe o *frameworks* –marcs de treball– de persistència).

- **Entitats JPA**. Objectes Java amb capacitats de ser persistits en una base de dades relacional. Aquests objectes han de seguir un conjunt de regles definides per l'estàndard JPA[14].

(14)En l'especificació EJB 3.x se substitueixen els EJB d'entitat per entitats JPA.

***Frameworks* de persistència**

Els marcs de treball de persistència són sistemes que automatitzen les operacions **CRUD** (creació, lectura, actualització i esborrament) d'entitats en les bases de dades relacionals. Aquests *frameworks* poden arribar a ser molt complets i convertir-se en vertaderes eines de mapatge objecte-relació.

Si volem accedir a fonts de dades que no siguin bases de dades relacionals haurem de fer servir JCA[15] o bé JDO[16]. Aquests tipus d'accés queden fora de l'abast dels materials.

(15)De l'anglès *Java Connector Architecture*.

(16)De l'anglès *Java Data Objects*.

Capa de serveis transversals

La **capa de serveis transversals** proporciona accés als components de qualsevol capa a un conjunt de serveis comuns que poden necessitar tots els components (per exemple, serveis d'autenticació, autorització, registre, instrumentació, etc.).

1.2.4. Distribució física

Podeu veure els nivells com la distribució física (en màquines) dels components de les capes lògiques que formen part de l'aplicació. Cal tenir en compte que la separació de les capes lògiques en nivells físics separats impacta tant en el rendiment com en l'escalabilitat de l'aplicació. Cal anar en compte a l'hora de definir aquestes separacions.

Bibliografia complementària

Teniu una descripció acurada dels nivells físics de desplegament al llibre C. de la Torre; U. Zorrilla; J. Calvarro; M. A. Ramos (2010). *Guía de Arquitectura* N-*Capas Orientada al Dominio con .NET 4.0*. Madrid: Krassis Consulting.

Hi ha diferents patrons de distribució física basada en nivells; a continuació esmentarem els més habituals per a aplicacions Java EE.

1) Arquitectura de 2 nivells (client-servidor)

Escenari típic d'aplicacions client-servidor. Hi ha un nivell client que conté totes les capes lògiques de l'aplicació i es comunica directament amb la capa EIS (bàsicament amb servidors de bases de dades). Aquest escenari no és gens habitual en aplicacions Java EE.

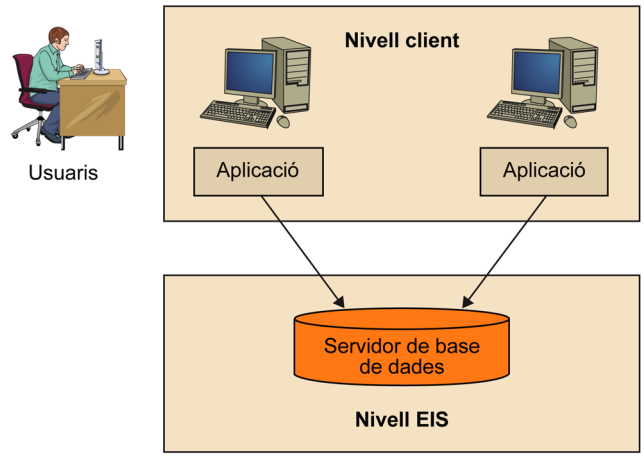

Figura 5. Arquitectura de 2 nivells (client-servidor)

2) Arquitectura de 3 nivells

En aquest escenari la capa de presentació pot estar en el segon nivell per al cas d'aplicacions web o bé en el primer nivell en el cas de miniaplicacions i aplicacions d'escriptori; la capa de negoci i la capa d'integració estaran en el segon nivell i la capa EIS estaria en un tercer nivell. Aquest és l'escenari més típic per a aplicacions JEE en què no cal separar físicament el servidor web del servidor d'aplicacions per raons de seguretat ni de càrrega.

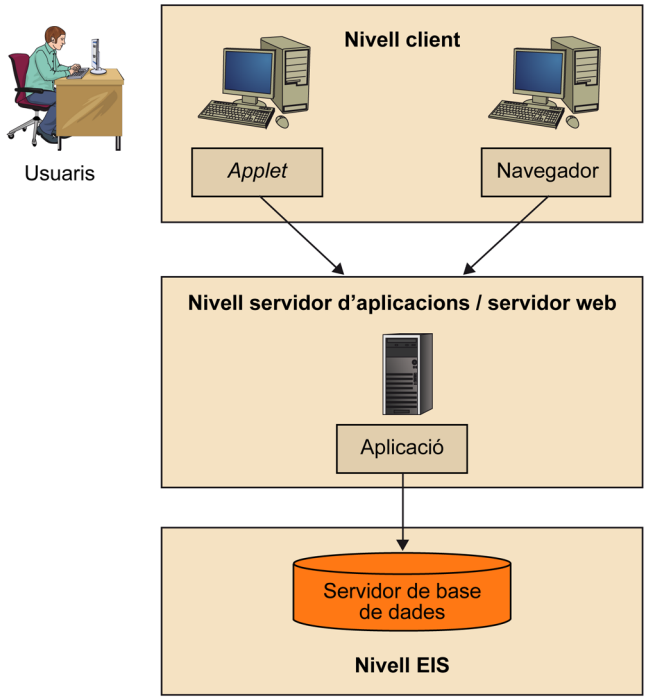

Figura 6. Arquitectura de 3 nivells

3) Arquitectura de *n* nivells

Aquest escenari és idèntic a l'anterior, però separant físicament la capa de presentació de la capa de negoci i integració. En aquest escenari cal separar la capa de presentació de la de negoci per raons de seguretat i, normalment, es col·loquen en xarxes diferents i amb tallafocs entre aquestes.

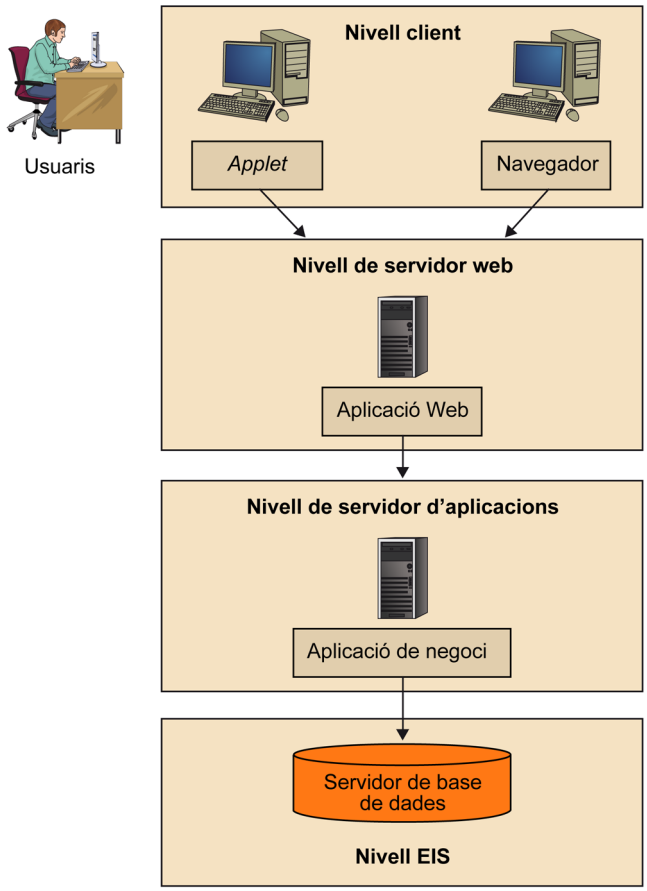

Figura 7. Arquitectura de 4 nivells

Com a resum, en aquest apartat hem vist que les aplicacions Java EE s'emmarquen en un estil arquitectònic multicapa amb *n* nivells, hem fet una breu descripció de les capes en què podem dividir l'aplicació i dels components que podem trobar a cadascuna.

En l'apartat següent analitzarem el cicle de vida d'una aplicació Java EE, fixant-nos especialment en els processos d'assemblatge (empaquetament) i desplegament de l'aplicació i en els rols que intervenen en el cicle de vida d'un desenvolupament.

1.2.5. Cicle de vida

Un cop desenvolupats els components que formaran l'aplicació Java EE s'han d'**empaquetar** i **desplegar** en el servidor d'aplicacions perquè els clients els puguin fer servir.

Per a desplegar-los, primer cal crear un mòdul empaquetant els components desenvolupats amb els fitxers relacionats i, opcionalment, un descriptor de desplegament del component.

Anotacions

A partir de la versió 5 els descriptors de desplegament passen a ser opcional, ja que moltes de les tasques de configuració es poden fer amb anotacions.

Les **anotacions** permeten afegir metadades al codi font, que estaran disponibles per a l'aplicació en temps d'execució.

Un **descriptor de desplegament** és un fitxer XML associat a un component, un mòdul o una aplicació Java EE que proporciona la informació necessària per a fer el desplegament en el servidor d'aplicacions. Indica al contenidor declarativament quin comportament tindran els components que formen el mòdul pel que fa a seguretat, transaccions, persistència, etc. El fet de tenir una configuració declarativa i no programàtica facilita la portabilitat dels components i mòduls.

Un **mòdul** és la unitat mínima que es pot desplegar en un servidor d'aplicacions. Tot i que els mòduls es poden desplegar sense formar part d'una aplicació, l'habitual és ajuntar aquests mòduls per a crear una aplicació Java EE.

Per a desplegar una aplicació Java EE cal, doncs, empaquetar tots els mòduls que formen l'aplicació i crear un descriptor de desplegament per a l'aplicació que enumeri tots els mòduls que en formen part.

Una aplicació Java EE està formada per un o més mòduls i un descriptor de desplegament empaquetats amb una estructura determinada. Els mòduls són les unitats bàsiques de construcció d'aplicacions Java EE.

La figura 8 esquematitza com es poden estructurar els components, mòduls i aplicacions Java EE per a desplegar-los en el servidor d'aplicacions.

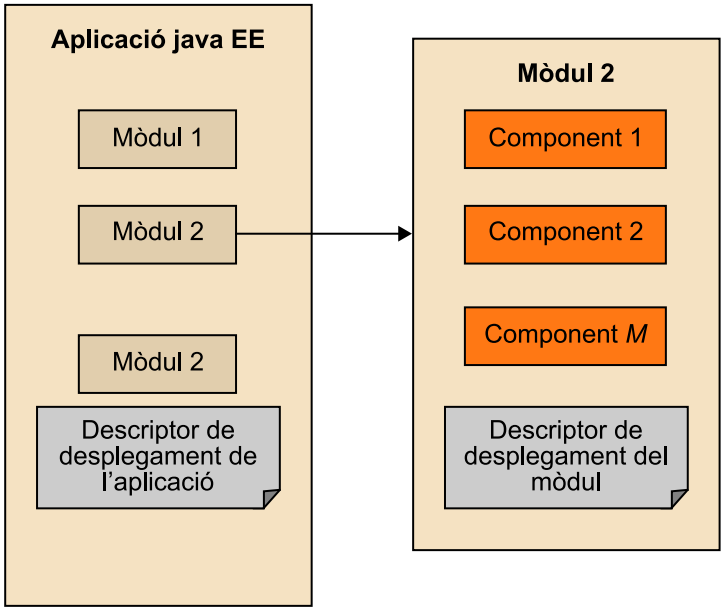

Figura 8. Components, mòduls i aplicacions Java EE

El cicle de vida típic per a una aplicació Java EE comença amb la creació dels components que la formaran i l'empaquetament d'aquests per a formar els mòduls. L'aplicació està formada per diversos mòduls que s'ajunten en el procés d'empaquetament i s'instal·len en el servidor d'aplicacions en el procés de desplegament.

En la figura 9 teniu el cicle de vida complet d'una aplicació Java EE.

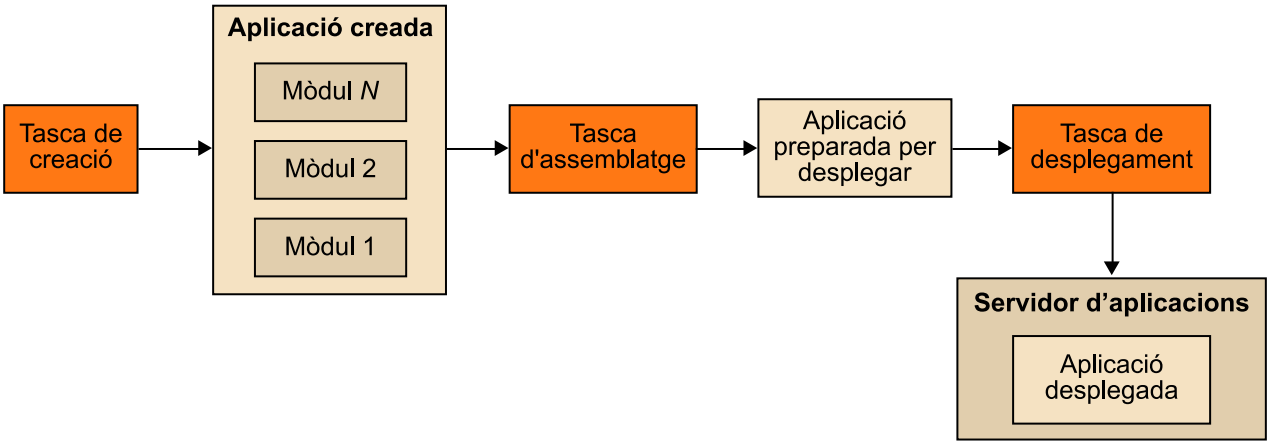

Figura 9. Cicle de vida complet d'una aplicació Java EE

A continuació veurem amb una mica més de detall cadascuna d'aquestes fases.

Creació

En la **fase de creació** s'agafen els components desenvolupats, es crea el fitxer de desplegament per a cada component i s'empaqueten amb una estructura determinada per a produir el mòdul.

Eines per a la creació

Actualment hi ha eines que faciliten tant la creació dels descriptors de desplegament dels components com l'empaquetament per a formar els mòduls i les aplicacions.

Totes les especificacions de components de la plataforma Java EE inclouen en la seva definició l'estructura que cal seguir per a empaquetar-los i desplegar-los en els contenidors, i també el format (esquema XML) que ha de tenir el fitxer de desplegament del mòdul.

La sortida del procés de creació és un mòdul, la unitat bàsica a partir de la qual es formen les aplicacions Java EE.

Java EE defineix quatre tipus de mòduls:

Mòduls

Queda fora de l'abast d'aquests materials l'estructura de fitxers concrets de cada tipus de mòdul i la definició del seu descriptor de desplegament. Podeu trobar aquesta informació en l'especificació de la plataforma Java EE.

1) Mòdul client. Fitxer amb extensió *.jar* que conté les classes de l'aplicació client, el descriptor de desplegament i els fitxers relacionats.

2) Mòdul web. Fitxer amb extensió *.war* que conté els components web (*servlets* i pàgines web), el descriptor de desplegament i els recursos relacionats (pàgines HTML, JavaScript, imatges, etc.).

3) Mòdul EJB. Fitxer amb extensió *.jar* que conté tots els fitxers del component EJB, el seu descriptor de desplegament i els fitxers relacionats.

4) Mòdul d'adaptadors de recursos. Fitxer amb extensió *.rar* que conté els adaptadors de recursos, el seu descriptor de desplegament i els fitxers relacionats. Aquests components no els veurem en aquest curs.

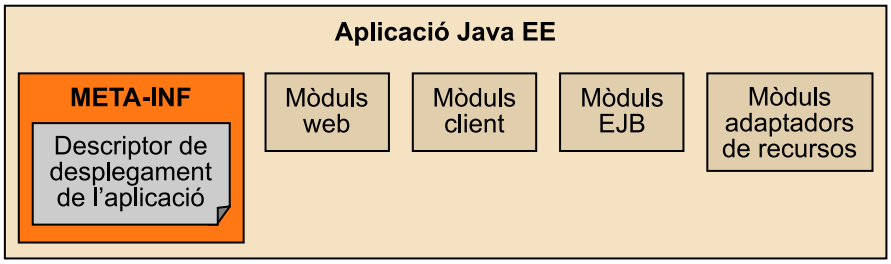

Figura 10. Estructura d'una aplicació Java EE

Assemblatge

La **fase d'assemblatge** crea el fitxer amb l'aplicació que es vol desplegar en el servidor d'aplicacions.

Els mòduls creats en la fase de creació es poden desplegar directament en el servidor d'aplicacions o bé combinar-los per a crear una aplicació Java EE.

Una aplicació Java EE està formada per un conjunt de mòduls i un descriptor de desplegament de l'aplicació. S'empaqueta en un fitxer amb extensió *.ear* per a formar una unitat de desplegament.

En la fase d'assemblatge de l'aplicació hem de generar el descriptor de desplegament de l'aplicació i crear el fitxer *.ear* amb els mòduls que formen l'aplicació.

Desplegament

La **fase de desplegament** instal·la i configura l'aplicació Java EE en el servidor d'aplicacions.

Per a fer la instal·lació de l'aplicació, el servidor d'aplicacions llegeix el descriptor de desplegament i desplega cada component enumerat en el contenidor que li toca (els mòduls EJB en el contenidor dels EJB, els mòduls web en el contenidor web, etc.), i assigna als components de cada mòdul les propietats que tenen configurades en els descriptors de desplegament respectius.

Procés de desplegament

Podeu veure el procés de desplegament d'un component com un procés d'instal·lació i configuració del component en el servidor d'aplicacions.

Els servidors d'aplicacions solen proporcionar eines per a fer el desplegament d'aplicacions.

Mecanismes de desplegament

L'estàndard no diu res del mecanisme que cada servidor d'aplicacions ha de proporcionar per a fer el desplegament de les aplicacions. Alguns servidors proporcionen una interfície gràfica per a fer aquesta tasca, mentre que en altres aquest procés és tan simple com copiar el fitxer *.ear* de l'aplicació a un directori concret del servidor d'aplicacions.

1.2.6. Rols en el desenvolupament

En l'apartat anterior hem analitzat quins passos calien per a desenvolupar una aplicació Java EE. Aquest procés i l'estructura eminentment modular i basada en components de l'arquitectura Java EE permet repartir el desenvolupament i el desplegament de l'aplicació en diferents rols que duen a terme diverses tasques en el procés.

- **Proveïdor d'eines**. Proporciona eines per al desenvolupament i el desplegament d'aplicacions Java EE. Aquestes eines poden ser IDE de desenvolupament, entorns d'anàlisi i disseny en UML, entorns de test, etc.

- **Proveïdors de components**. Són els desenvolupadors dels components de les aplicacions. N'hi ha de molts tipus: des de dissenyadors que fan les JSP de la capa de presentació fins a experts en el negoci que desenvolupen components de negoci específics per a cada domini particular. Els proveïdors de components també defineixen els descriptors de desplegament

dels components i els empaqueten. Podem veure els proveïdors de components com els desenvolupadors dels components que formaran l'aplicació.

- **Muntador d'aplicacions**. Combina els mòduls amb els diferents components per a crear l'aplicació sencera que s'ha de desplegar en el servidor d'aplicacions. També és responsable de crear el descriptor de desplegament de l'aplicació i de donar la informació que el desplegador necessita per a fer el desplegament.

- **Proveïdor dels contenidors i del servidor d'aplicacions**. Proporciona una implementació compatible amb els estàndards que defineix la plataforma Java EE i incorpora implementacions de les API, serveis i contenidors que defineix l'estàndard. Són els fabricants dels servidors d'aplicacions com, per exemple, Oracle (que fabrica WebLogic) o Jboss.

- **Desplegador d'aplicacions**. S'encarrega de fer la instal·lació d'una aplicació o un mòdul Java EE amb les eines que li proporciona cada servidor d'aplicacions, de configurar tota la informació de l'entorn que necessita l'aplicació i posar en marxa l'aplicació en el servidor. Durant la configuració, el desplegador segueix instruccions dels proveïdors dels components per a poder resoldre dependències externes, especificar variables de seguretat i assignar atributs de transaccions. Durant la instal·lació, el desplegador trasllada els components de l'aplicació al servidor i genera classes i interfícies específiques del contenidor on es fa el desplegament (això ho fa automàticament l'eina de desplegament del servidor d'aplicacions).

- **Administrador de sistemes**. Pel que fa a l'entorn Java EE, l'administrador de sistemes s'encarrega d'administrar i monitorar els servidors.

La figura 11 mostra com es relacionen els rols anteriors amb les tasques que fa cadascun.

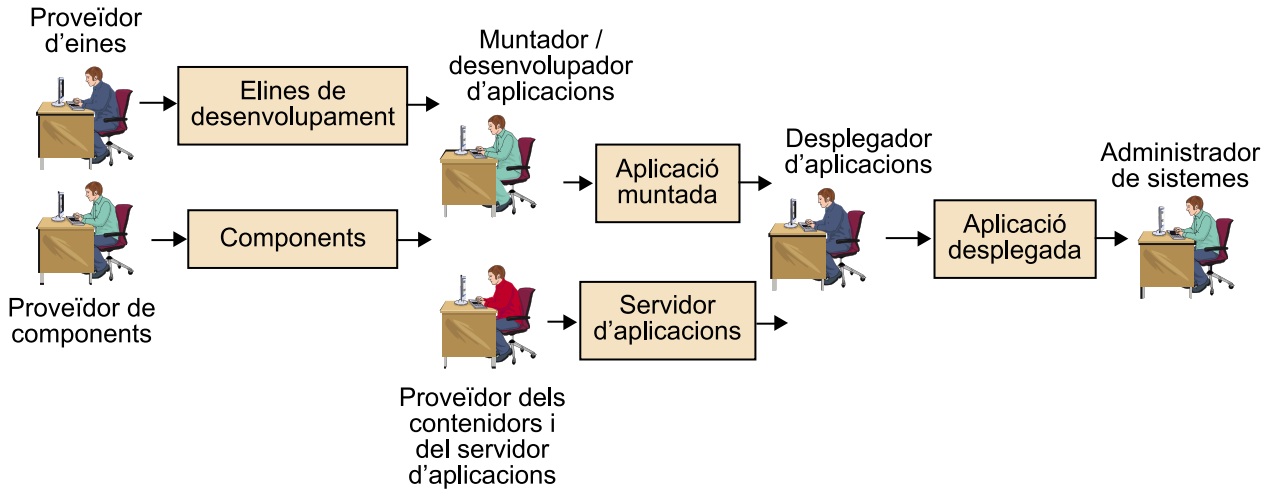

Figura 11. Rols i tasques

En organitzacions grans és habitual que es diferenciïn tots aquests rols, però en organitzacions petites alguns es fusionen. És freqüent que els rols de muntador d'aplicacions i de proveïdor de components es fusionin en un rol de **desenvolupador**, i els d'administrador de sistemes i de desplegador d'aplicacions, en un rol d'**administrador**.

1.3. Plataforma Java EE

En l'apartat anterior hem vist amb detall el model de programació d'aplicacions que defineix Java EE i l'estil arquitectònic en què s'emmarquen aquestes aplicacions. En aquest apartat veurem molt per damunt algunes de les API, serveis i tecnologies que proporciona Java EE.

1.3.1. Serveis i tecnologies Java EE

En analitzar el model de components i contenidors vam veure que els contenidors donaven als components un conjunt de serveis. Aquests serveis simplifiquen el desenvolupament d'aplicacions i en permeten la personalització en temps de desplegament.

El conjunt d'API, serveis i tecnologies que ha de suportar una determinada versió de la plataforma Java EE es recull en l'especificació formal de la plataforma que defineix Oracle.

Web recomanat

Teniu l'especificació formal de les diferents versions de la plataforma Java EE a la web d'Oracle http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html.

En aquest apartat veurem a grans trets quins serveis i tecnologies ofereix la plataforma Java EE i les API estàndard en què es basen aquests serveis. Queda fora de l'abast d'aquests materials fer una anàlisi exhaustiva de totes les API i tecnologies que proporciona Java EE.

Serveis, API i tecnologies

Els serveis, les API i les tecnologies recollides en aquest material es refereixen a la versió 6 de l'especificació de Java EE.

Pel que fa a serveis, Java EE ens proporciona, entre d'altres, els següents:

- **Servei de noms**. Permet l'accés a recursos i components mitjançant una API unificada d'accés (JNDI). Quan un component o un recurs es desplega en el servidor d'aplicacions, el servei de desplegament l'ubica a l'arbre JNDI del servidor perquè els clients hi puguin accedir. Els clients fan servir el servei de noms i directoris per a accedir als recursos i components desplegats.

- **Servei de transaccions**. Aquest servei fa que els desenvolupadors no hagin d'escriure codi per a gestionar el comportament transaccional dels components. L'especificació del comportament transaccional s'efectua en el descriptor de desplegament i es configura quan s'assembla el component.

- **Servei de seguretat**. Assegura que només accedeixin als components i recursos aquells que hi tinguin accés. El servei de seguretat també proporciona mecanismes d'autenticació i de confidencialitat.

- **Servei de connectivitat remota**. Gestiona les comunicacions a baix nivell entre components remots, i fa que el fet que les crides siguin remotes sigui transparent per als clients.

- **Serveis de desplegament**. Abans que un component es pugui executar, s'ha de muntar i desplegar al seu contenidor. El servei de desplegament configura i "instal·la" els components en els contenidors i habilita l'accés a les aplicacions.

Amb el procés de desplegament es configuren atributs (per exemple, els atributs de seguretat i transaccionals) dels components perquè el contenidor pugui implementar la funcionalitat requerida en cada cas. El desenvolupador configura declarativament aquests atributs en un fitxer XML anomenat *descriptor de desplegament* o bé amb anotacions en el codi dels components.

Per exemple, en el descriptor de desplegament d'un component podem especificar que només els usuaris que tinguin el rol d'administradors hi puguin accedir; llavors el contenidor s'encarregarà de limitar l'accés a aquest component. L'important és que això es fa en temps de desplegament, i no es troba codificat en l'aplicació.

El servei de desplegament permet canvis en el comportament i la configuració d'un component o bé d'una aplicació sense haver de canviar el codi font.

Pel que fa a les API l'important és tenir en compte que una determinada especificació de Java EE suporta un conjunt d'API amb la seva versió corresponent. D'aquesta manera, assegurem que un desenvolupament que faci servir una API suportada per l'especificació no tindrà cap problema per a executar-se en un servidor d'aplicacions que compleixi amb l'especificació.

Més informació

Teniu tota la informació de les API i les tecnologies de cada versió de Java EE a la web d'Oracle.

El contenidor també gestiona serveis no configurables com els cicles de vida dels components, les fonts de recursos de connexió a les bases de dades, la persistència de dades i l'accés a les API de Java EE.

1.3.2. Tests de compatibilitat

Com que l'especificació de Java EE està regida per un JSR[17], ha de publicar uns tests de compatibilitat perquè tots els fabricants de servidors d'aplicacions puguin certificar que els seus productes compleixen amb el que diu l'especificació. Aquest conjunt de tests s'anomena *CTS*[18].

(17)De l'anglès *Java Specification Request*.

(18)De l'anglès *Compatibility Test Suite*.

El **CTS** és un conjunt de tests (més de 15.000) definits per Oracle que formen part de Java EE i que els fabricants de servidors d'aplicacions han de passar per a certificar que el servidor compleix amb el que diu l'estàndard i poder penjar l'etiqueta de *Java EE compliant*.

L'objectiu del CTS és assegurar que els productes (servidors d'aplicacions en el cas de Java EE) que es desenvolupen per a suportar aplicacions Java EE tinguin un mínim de qualitat, compleixin amb l'especificació i proporcionin tots els serveis obligatoris que exigeix la plataforma.

Quan un servidor d'aplicacions passa els tests de compatibilitat obté una certificació de compatibilitat que l'acredita i pot presentar per a assegurar que els usuaris obtenen les funcionalitats esperades amb el producte.

Un servidor d'aplicacions que hagi passat aquests tests ha de ser capaç de desplegar i executar qualsevol aplicació que compleixi amb les especificacions de Java EE. Mirant això des d'un altre punt de vista, també podem dir que una aplicació que compleixi amb les especificacions de Java EE ha de poder ser desplegada i executada en qualsevol servidor d'aplicacions que hagi passat els tests de compatibilitat.

Això, juntament amb el fet que els components que formen les aplicacions Java EE estan escrits en Java, fa que les aplicacions siguin portables entre servidors d'aplicacions de diferents fabricants i, també, entre plataformes sense gairebé cap canvi.

1.3.3. Implementació de referència

Qualsevol especificació ha d'oferir una implementació de referència de la tecnologia que està especificant perquè els desenvolupadors tinguin algun producte amb el qual puguin provar que els seus desenvolupaments són compatibles amb l'especificació.

La **implementació de referència** de Java EE és un servidor d'aplicacions desenvolupat anomenat Oracle GlassFish Server, que es pot descarregar de la web d'Oracle i que serveix per a verificar que un desenvolupament compleix amb l'estàndard Java EE.

És molt recomanable provar l'aplicació que estem desenvolupant amb la implementació de referència per tal de certificar que el codi creat és estàndard i no fem servir biblioteques de tercers que farien que aquest codi no fos portable entre els diferents servidors d'aplicacions compatibles amb Java EE.

Implementació de referència

Amb la implementació de referència podreu fer proves de les darreres novetats de l'especificació, potser abans que ho pugueu fer amb els altres servidors d'aplicacions. Oracle avisa que aquesta implementació només s'ha de fer servir en desenvolupament, i no es recomana instal·lar-la en servidors de producció.

GlassFish

Amb l'especificació 6 de Java EE Oracle ofereix dues versions del servidor d'aplicacions GlassFish: una de completa i l'altra només amb el perfil web.

2. Arquitectura lògica

En el primer apartat del mòdul hem explicat de manera genèrica la plataforma Java EE, sense entrar-hi en massa detall. Hem vist que Java EE segueix un estil arquitectònic client-servidor multicapa amb *n* nivells i que les aplicacions es poden dividir, bàsicament, en quatre capes, que s'executaran en els contenidors del servidor d'aplicacions:

1) Capa de presentació
2) Capa de negoci
3) Capa d'integració
4) Capa de serveis transversals

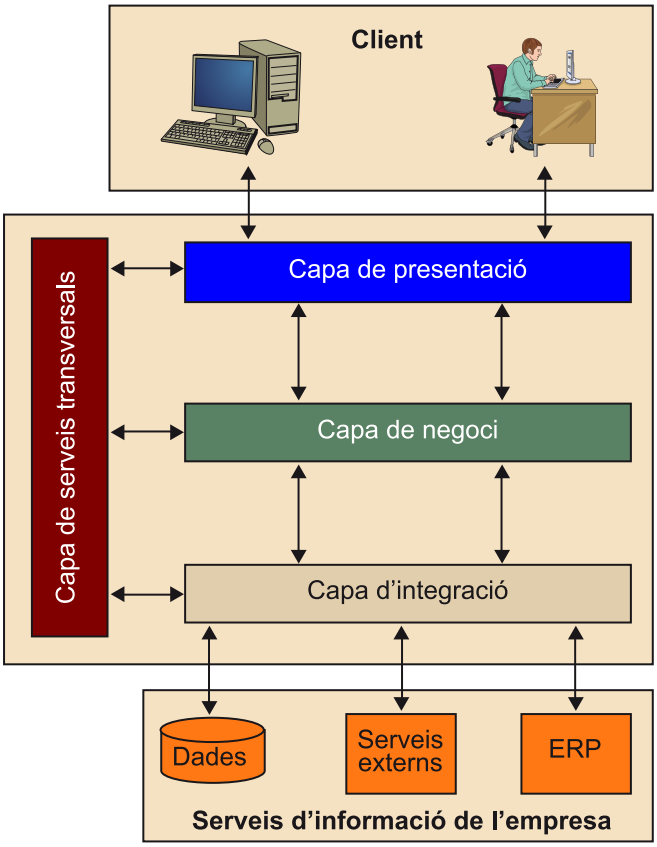

Figura 12. Arquitectura lògica d'un sistema multicapa

En aquest apartat tractarem amb profunditat cadascuna d'aquestes quatre capes: primer analitzarem la capa de presentació per a aplicacions web i veurem quins components defineix Java EE per a implementar aquesta capa, després analitzarem les capes de lògica de negoci, d'integració i de serveis transversals amb les propostes de Java EE per a implementar les dues primeres, i, finalment, veurem un parell d'exemples de distribució en nivells d'aquestes capes en aplicacions Java EE.

Per a totes les capes oferirem algunes recomanacions de disseny que cal tenir en compte a l'hora de dissenyar les aplicacions.

2.1. Capa de presentació

En aquest subapartat ens centrarem en la tecnologia i els components que defineix Java EE per a implementar la capa de presentació per a entorns web (a partir d'ara l'anomenarem ***capa web***).

Veurem quins objectius té la capa web d'una aplicació i quins components i tecnologies permeten assolir aquests objectius; analitzarem com s'empaqueta i es desplega una aplicació web, quines són les arquitectures més típiques en aquest tipus d'aplicacions i, finalment, us donarem algunes recomanacions a l'hora de fer el disseny d'aquesta capa.

2.1.1. Objectius

La **capa web** d'una aplicació Java EE té com a objectiu primordial fer accessible la lògica de negoci a clients que accedeixin a l'aplicació mitjançant el protocol HTTP (per exemple, amb un navegador).

La capa web d'una aplicació Java EE no és només la capa encarregada de donar format al contingut dinàmic que es vol enviar als clients; fa moltes altres tasques, entre les quals cal destacar:

- Permetre als clients cridar la lògica de negoci de l'aplicació recollint les peticions d'aquests, cridant el mètode de negoci corresponent i presentant els resultats als clients.
- Generar contingut dinàmic per als clients en diversos formats.
- Gestionar el flux de presentació, és a dir, donada una entrada i un resultat en l'execució de la lògica de negoci, decidir quina és la pantalla següent que cal mostrar a l'usuari.
- Fer certes validacions a les dades que rep com a entrada.
- Mantenir l'estat entre diferents peticions sobre el protocol HTTP (que no té estat).
- Donar suport a diversos tipus de clients. Per exemple, la capa web d'una aplicació Java EE podria servir contingut a clients lleugers i a dispositius mòbils.

- Pot implementar la lògica de negoci[19].

(19)Tot i que és possible implementar la lògica de negoci de l'aplicació en la capa de presentació, no és ni la pràctica més habitual ni la més recomanable.

L'especificació de Java EE defineix un conjunt de tecnologies i components que es poden utilitzar per a algun propòsit en la capa web, i que veurem en els apartats següents:

- Components *Java Beans*
- *Servlets*
- *Java Server Faces*
- *Java Server Facelets*
- Llenguatge d'expressions
- *Java Server Pages*
- *Java Server Pages Standard Tag Library*

HTTP

HTTP (*Hypertext Transfer Protocol*) és un protocol sense estat d'enviament i recepció d'informació a través d'Internet. La comunicació seguint aquest protocol implica un client (molts cops un navegador) i un servidor (servidor web).

[20]De l'anglès *Common Gateway Interface*.

2.1.2. Evolució

Inicialment, es feia servir Internet per a proporcionar contingut estàtic; hi havia un conjunt de servidors HTTP que rebien les peticions dels clients i els tornaven la informació en forma d'HTML estàtic.

Aviat es va veure el gran potencial d'Internet per a proporcionar als usuaris no solament contingut estàtic, sinó també contingut **dinàmic** i **personalitzat**. L'objectiu següent va ser aconseguir generar contingut dinàmic a partir d'una petició HTTP. Per a aconseguir-ho es van desenvolupar mòduls que estenien els servidors HTTP i podien generar contingut dinàmic anomenat ***CGI***[20].

CGI

Un dels punts febles dels CGI era el rendiment, ja que cada vegada que el servidor rebia una petició havia de crear un procés nou per a processar-la.

Això també fa que una solució basada en CGI sigui molt poc escalable.

Els CGI, tot i que van esdevenir un mecanisme molt utilitzat per a generar contingut dinàmic, tenen algun problemes importants com, per exemple, l'eficiència.

La plataforma Java EE i els **components web** sorgeixen com a alternativa als CGI. Es basen en l'ús de Java com a llenguatge per a desenvolupar les "extensions" i a utilitzar un contenidor per a proporcionar a aquestes extensions el suport als serveis de sistema.

Els components web solucionen els problemes de rendiment dels CGI, són portables i poden accedir a molts serveis proporcionats pel contenidor en el qual es despleguen.

HTTP i el paradigma petició-resposta

Moltes vegades la capa web d'una aplicació Java EE està desenvolupada sobre el protocol HTTP seguint el paradigma de petició-resposta. Els components web reben peticions HTTP del servidor web, les processen i generen una resposta segons aquesta petició.

Hi ha un servidor en la capa web que rep les peticions HTTP dels clients; si aquestes van dirigides a un component web, les passa al contenidor web, que s'encarregarà de processar-les i tornar la resposta al client. Si la petició no va

dirigida a cap component web (per exemple, una petició d'una imatge, d'una pàgina HTML estàtica, una pel·lícula en Flash, etc.), el servidor web envia la resposta al client sense passar pel contenidor web.

2.1.3. *Servlets*

> Un *servlet* és un component web escrit en llenguatge Java que estén la funcionalitat d'un servidor web, rep peticions HTTP i genera contingut dinàmic com a resposta a aquestes peticions.

Els *servlets* "viuen" dins d'un contenidor web i es carreguen i executen dinàmicament com a resposta a les peticions que els clients fan a una URL determinada. Basen el seu funcionament en el paradigma de petició-resposta sota el protocol HTTP.

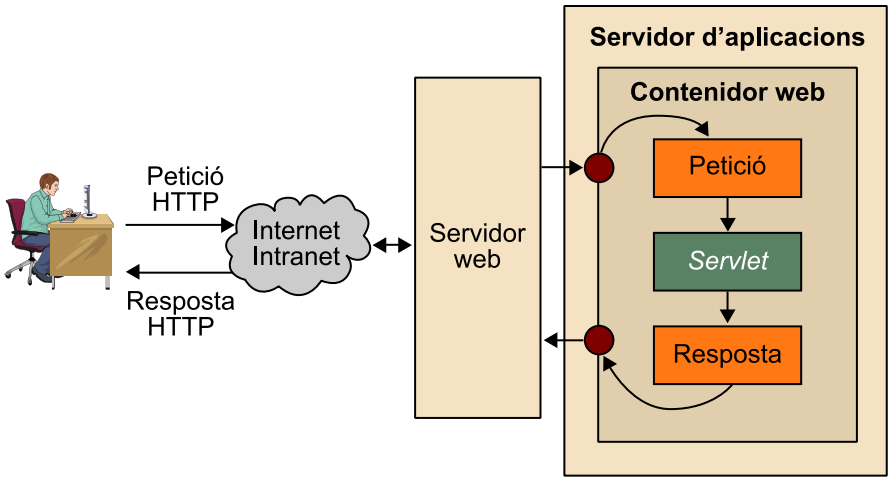

Figura 13. Paradigma de petició-resposta amb *servlets*

El servidor web i el contenidor de *servlets* es poden trobar a la mateixa màquina o bé en màquines diferents.

Els passos per a gestionar una petició HTTP són els següents:

1) El client fa una petició HTTP a una URL a la qual hi ha associada un *servlet*.

2) El servidor web rep aquesta petició, s'adona que està dirigida a un *servlet* i passa la petició al contenidor de *servlets*.

3) El contenidor de *servlets* crea un objecte Java que representa la petició i passa la petició al *servlet* cridant el mètode *service()* d'aquest.

4) El *servlet* processa la petició:

- Analitza els paràmetres de la petició.
- Executa les tasques que requereixi per a completar la petició.

- Crea una resposta (molts cops la resposta és una pàgina HTML).

5) El *servlet* torna la resposta al contenidor de *servlets*.

6) El contenidor de *servlets* envia la resposta al servidor web.

7) El servidor web l'envia al client via HTTP.

A continuació es presenta un exemple de *servlet* que mostra una pàgina HTML que diu "Hola!".

Cicle de vida dels components

Recordeu que la gestió del cicle de vida dels components era un servei que proporcionaven els contenidors als components que tenen desplegats.

```
public class ServletSimple extends HttpServlet {
public void service
  (HttpServletRequest req, HttpServletResponse res)
throws IOException, ServletException
  {
res.setContentType("text/html");
PrintWriter out = res.getWriter();
out.println("<HTML><HEAD>");
out.println("<TITLE>Hola</TITLE>");
out.println("</HEAD>");
out.println("<BODY>Hola!</BODY></HTML>");
out.close();
  }
}
```

Fixeu-vos que el *servlet* genera el contingut dinàmic amb codi Java, que escriu directament el codi HTML que s'enviarà al client.

Aquesta manera de generar l'aspecte gràfic de les aplicacions té diversos problemes greus:

- Fa molt complex crear la presentació, ja que no es poden utilitzar eines de disseny gràfic per a donar aspecte a les pàgines HTML que rep el client. No podem fer servir editors WYSIWYG per a crear la capa de presentació i es fa molt difícil fer el disseny gràfic sofisticat que requereixen la majoria d'aplicacions.

WYSIWYG

WYSIWYG és l'acrònim de l'anglès *What You See Is What You Get* ('el que veus és el que obtens'), i es pot aplicar a editors d'HTML per a indicar que podem escriure un document veient-ne l'aspecte final.

- El cost de manteniment és molt alt, més encara si tenim en compte que l'aspecte gràfic de les aplicacions serà el component que més canvia al llarg del temps.

- No promou una separació clara de rols en el desenvolupament. Els mateixos desenvolupadors que programen amb Java la lògica de presentació han de crear l'HTML que doni l'aspecte gràfic. Per als dissenyadors gràfics, és difícil generar l'aspecte de l'aplicació programant en Java.

2.1.4. *Java Server Faces*

Java Server Faces és un *framework* de components de servidor que implementa el patró MVC i serveix per a construir aplicacions web amb Java. JSF és la tecnologia "oficial" de la plataforma Java EE per a la capa web.

Les aplicacions JSF "viuen" dins d'un contenidor web i basen el seu funcionament en el paradigma petició-resposta sota el protocol HTTP.

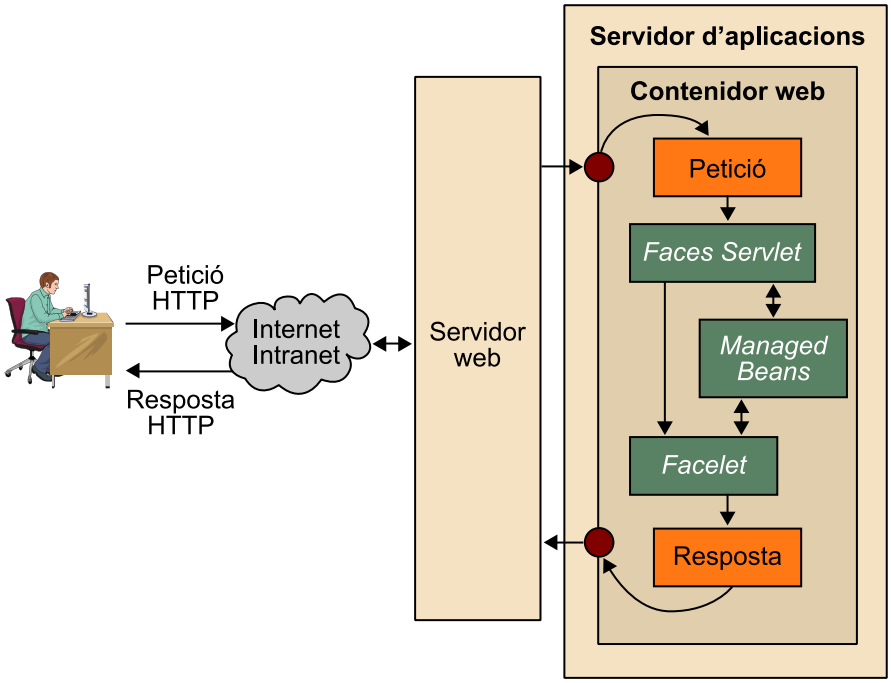

Figura 14. Paradigma de petició-resposta amb *Java server faces*

La tecnologia *Java Server Faces* està formada per:

- Una API per a representar components i gestionar-ne l'estat, gestionar esdeveniments, fer validacions de servidor, fer conversions de dades, definir la navegació entre pàgines, suport per a internacionalització i accessibilitat i mecanismes d'extensió per a aquestes característiques.

- Un conjunt de biblioteques d'etiquetes per a afegir components a les pàgines web i connectar-los a objectes de servidor.

Com a implementació del patró MVC *Java Server Faces* consta de vistes, model i controlador. Analitzem com s'implementa cada una d'aquestes parts:

1) Les **vistes** proporcionen la interfície d'usuari de l'aplicació web:

- Les vistes d'una aplicació JSF s'implementen amb llenguatges de declaració de pàgines, inicialment es feia servir JSP[21] i, actualment, el més habitual és utilitzar *facelets*.

- JSF defineix un conjunt de biblioteques d'etiquetes per a implementar components d'interfície d'usuari. Per exemple, si volem posar un botó en una pàgina podem utilitzar l'etiqueta `<h:commandButton"/>`.

- Les vistes permeten vincular els components d'interfície d'usuari amb el model, mitjançant un llenguatge d'expressions. Aquesta vinculació es fa mitjançant esdeveniments.

[21] De l'anglès *Java Server Pages*.

Els ***facelets*** són el sistema per defecte per a crear vistes amb JSF. Són documents similars a una pàgina HTML, però que poden contenir etiquetes de diferents tipus (pròpies de *facelets*, etiquetes de JSF, etiquetes de JSTL, etc.) per a generar components d'interfície d'usuari i fer altres tasques. També tenen suport per al llenguatge d'expressions, un mecanisme molt potent per a comunicar les vistes amb el model, i un sistema de plantilles per a crear pàgines complexes.

JSTL

JSTL és l'acrònim de l'anglès *JSP Standard Tag Library*, i és un conjunt de biblioteques d'etiquetes simples i estàndard que encapsulen funcionalitat utilitzada habitualment per a escriure pàgines JSP.

2) El **model** és el responsable de la lògica de l'aplicació:

- El model està implementat amb *Managed Beans*, objectes Java gestionats pel *framework* de JSF i que s'encarreguen d'implementar la lògica de negoci, respondre als esdeveniments generats pels components de la interfície d'usuari i controlar la navegació entre pàgines amb els mètodes d'acció.

- Els *Managed Beans* poden implementar la lògica de negoci directament o bé delegar en components de la capa de negoci.

Managed Beans

Els *Managed Beans* són *Java Beans*, és a dir, classes Java normals que han de tenir un constructor públic sense paràmetres i accés a les propietats amb mètodes de l'estil *getXXX* i *setXXX*.

No els confongueu amb els *enterprise Java Beans*, que veurem en la capa de negoci.

3) El **controlador** s'encarrega de gestionar la lògica de presentació:

- El controlador està implementat amb un *servlet* anomenat *Faces Servlet* i un conjunt de mètodes d'acció dels *Managed Beans*.

- Totes les peticions HTTP de l'usuari les recull el *Faces Servlet*, que s'encarrega d'actualitzar la interfície d'usuari, les dades dels *Managed Beans* i cridar els gestors d'esdeveniments i els mètodes d'acció.

2.1.5. *Java Server Pages*

Una **JSP** és un document similar a una pàgina HTML però que, a més de generar contingut estàtic, pot incloure elements per a generar contingut dinàmic, com a resposta a una petició HTTP.

Els elements per a generar contingut dinàmic són bocins de codi escrits en Java i unes etiquetes del mateix tipus que les etiquetes HTML que interactuen amb objectes Java del servidor.

La tecnologia JSP té totes les capacitats dinàmiques dels *servlets*, però proporciona una manera més natural de crear el contingut estàtic.

Per exemple, el tros de codi següent és una JSP que mostra una pàgina HTML que diu "Hola!". És el mateix exemple que hem vist amb *servlets*, però escrit amb una JSP.

```
<html>
   <head>
      <title>Hola</title>
   </head>
   <body>
     Hola!
   </body>
</html>
```

En l'exemple anterior només es generava codi estàtic. En canvi, les JSP també són capaces de generar codi dinàmic. Així, per exemple, la JSP següent també mostrarà la data i l'hora de cada petició rebuda.

```
<html>
   <head>
      <title>Hola</title>
   </head>
   <body>
     Hola!
     <br>
     La data actual és: <%= java.util.Date() %>
   </body>
</html>
```

La tecnologia de *servlets* i la de JSP estan molt relacionades; de fet, les JSP són transformades internament a *servlets* quan es despleguen en el contenidor web. Per això, el seu cicle de vida és molt similar al que hem estudiat en el cas dels *servlets*; la diferència fonamental es troba en la sintaxi.

2.1.6. Evolució de les arquitectures web

L'arquitectura de les aplicacions web ha evolucionat força des del seu inici.

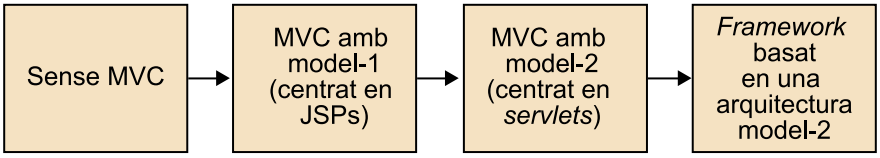

Figura 15. Evolució de les arquitectures web

Inicialment, la majoria d'aplicacions web mostraven contingut estàtic i les que necessitaven generar contingut dinàmic ho feien directament amb CGI o bé amb *servlets* o JSP. Aquesta manera de procedir feia aplicacions molt difícils de mantenir i no proporcionava cap tipus de reutilització.

Per a solucionar-ho es va passar a utilitzar algunes característiques de les JSP, que permetien fer una aplicació més estructurada i una mica més reutilitzable. Aquesta arquitectura s'anomena *Model-1* (vegeu la figura 16).

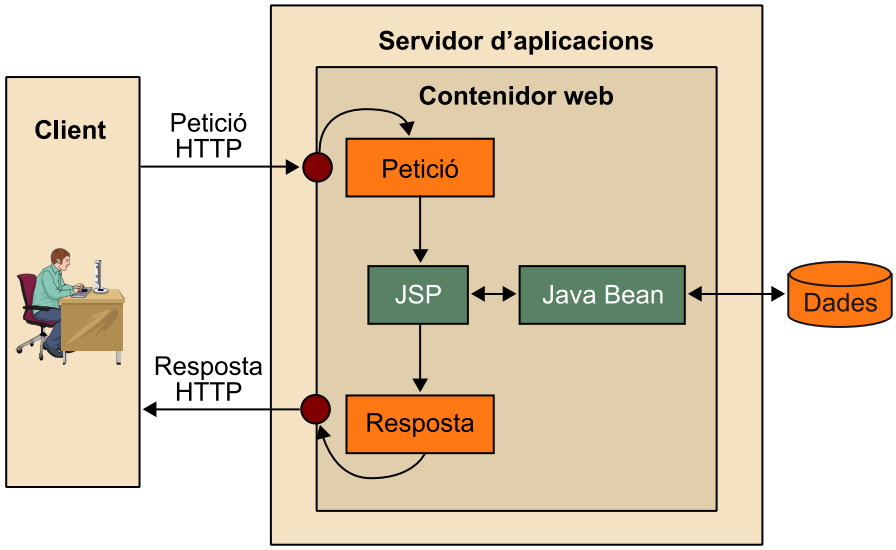

Figura 16. Arquitectura Model-1

En una arquitectura Model-1 els clients accedeixen directament a pàgines JSP. Aquestes fan servir *Java beans* per a executar la lògica de negoci (model de l'aplicació) i determinen elles mateixes la pàgina següent per mostrar.

L'arquitectura Model-1 té alguns desavantatges importants:

- La selecció de la vista següent està descentralitzada.
- Cada pàgina JSP és responsable de processar els paràmetres que li arriben.
- És difícil mostrar diferents pàgines depenent dels paràmetres d'entrada de la petició.

Això ens porta a definir una arquitectura anomenada *Model-2*.

En una arquitectura Model-2 els *servlets* i les JSP col·laboren per a implementar la capa de presentació de les aplicacions web. Hi ha un *servlet* "controlador" entre les peticions dels clients i les pàgines JSP que mostren la sortida.

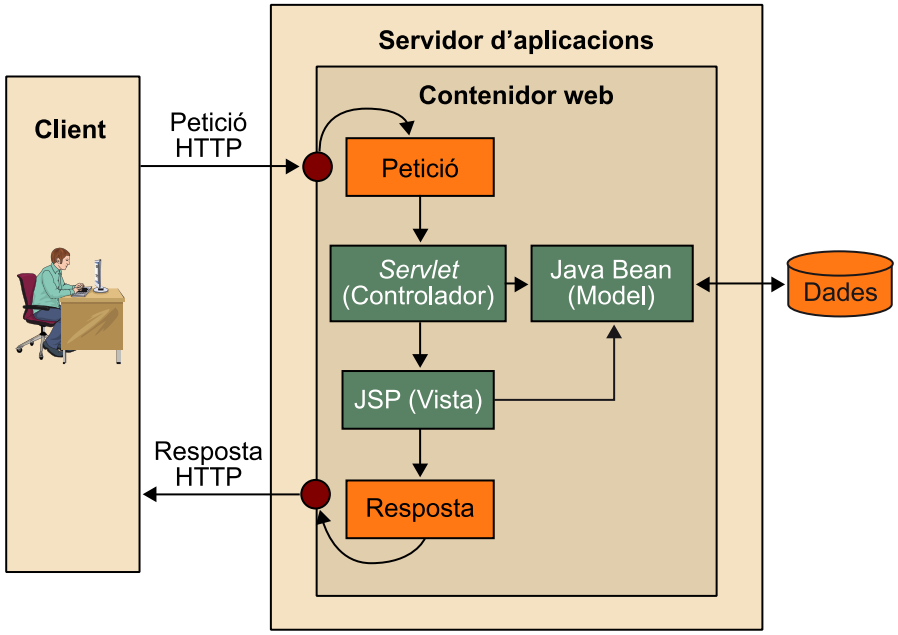

Figura 17. Arquitectura Model-2

Aquest controlador s'encarrega de centralitzar la lògica de selecció de la vista següent segons els paràmetres d'entrada, el resultat de la incoació a la lògica de negoci o l'estat de l'aplicació. El controlador també s'encarrega d'executar la crida a la lògica de negoci adient segons la petició que li arribi.

Les aplicacions basades en Model-2 permeten una separació clara de rols en la capa de presentació, són més flexibles, mantenibles i permeten tractaments centralitzats per a totes les peticions (per exemple, aplicar seguretat i registres). Per tot això, és recomanable seguir aquesta arquitectura per a la capa de presentació.

Si seguim una arquitectura basada en Model-2, hi ha un conjunt de tasques repetitives que s'han d'implementar per a totes les aplicacions com, per exemple, rebre els paràmetres d'entrada de les peticions i fer-hi validacions, cridar la lògica de negoci, triar la vista següent per mostrar, etc. Aquestes tasques es poden implementar en un *framework* per a la capa web que facin servir tots els desenvolupaments i d'aquesta manera no s'hagin d'implementar per a cada aplicació.

Els desenvolupadors crearan la capa web usant o estenent les classes i interfícies del *framework*. L'ús d'un *framework* per a la capa web basat en una **arquitectura Model-2** us proporciona els avantatges següents:

- Desacobla la capa de presentació de la capa de negoci en components separats.
- Simplifica i estandarditza la validació dels paràmetres d'entrada.
- Simplifica la gestió del flux de navegació de l'aplicació.
- Proporciona un punt central de control.
- Permet un nivell molt alt de reutilització.

- Imposa la mateixa arquitectura per a tots els desenvolupaments.
- Simplifica moltes tasques repetitives.

Hi ha molts marcs de treball per a la capa de presentació de les aplicacions web. Alguns són:

- Struts.
- WebWork.
- *Java Server Faces*.
- Spring (tot i que Spring és més genèric, també té un *framework* de presentació).

Tal com ja hem dit anteriorment, *Java Server Faces* és el *framework* recomanat per Java EE per a implementar la capa de presentació.

2.1.7. Recomanacions de disseny de la capa de presentació

Per a passar de l'especificació independent de la tecnologia que hem vist en mòduls anteriors a la implementació amb una tecnologia de components determinada (en aquest cas Java EE), cal fer una fase prèvia de disseny per tal d'evitar errors i reduir la distància existent entre l'especificació i la implementació concreta.

En aquesta fase de disseny es prenen decisions d'arquitectura que depenen de cada tecnologia concreta. En aquest subapartat farem una revisió breu de les arquitectures més habituals que podeu trobar en la capa web d'una aplicació Java EE. També us donarem algunes recomanacions de disseny per a fer el pas de l'especificació a la implementació de la capa de presentació per a les aplicacions web.

Més informació

Teniu informació detallada dels patrons de disseny més habituals de l'arquitectura Java EE per a la capa de presentació, segons proposa Sun Microsystems, al llibre:

D. Alur; **J. Crupi**; **D. Malks** (2003). *Core J2Ee Patterns: Best Practices and Design Strategies* (2a. Ed.). Upper Saddle River: Prentice Hall.

La capa de presentació és una de les capes que canvia més sovint. Un mal disseny pot fer que l'aplicació que desenvolupeu esdevingui molt complexa i poc mantenible.

Necessitem una aplicació web?

Tot i que, en aquest subapartat, ens hem centrat en els components que proporciona Java EE per a implementar la capa de presentació com una aplicació web, el primer que cal fer, sempre, és avaluar si realment una aplicació web és la millor alternativa per a implementar la capa de presentació.

Java EE **no** tanca la porta al desenvolupament de la capa de presentació amb tecnologies no basades en clients lleugers.

Una alternativa que cal considerar és fer la presentació amb una aplicació d'escriptori. Avalueu qüestions com:

- És necessari molta riquesa en la interfície gràfica, amb molts controls d'usuari complexos i interrelacionats?
- L'aplicació ha de donar una resposta molt ràpida a les interaccions de l'usuari?
- L'aplicació que s'ha de desenvolupar és accessible només a usuaris interns?
- Calen capacitats de processament fora de línia?
- La nostra aplicació ha d'interactuar amb altres aplicacions locals o bé amb el sistema operatiu de les màquines client?

Si la resposta a les qüestions anteriors és afirmativa, possiblement el disseny de la capa de presentació com una aplicació web i el client com a client lleuger no és la millor alternativa. En aquest cas, podeu considerar fer la presentació amb una aplicació d'escriptori (ja sigui amb Java o bé amb alguna altra tecnologia com .NET).

Smart client

Actualment, s'està parlant molt de *smart client*; les aplicacions de tipus *smart client* intenten crear clients que conjuguin el millor dels clients lleugers i les aplicacions d'escriptori.

Tingueu també en compte a l'hora de prendre una decisió si podeu utilitzar les noves característiques que ens aporten tant HTML5 com CSS3 en la implementació d'interfícies web. Moltes característiques que abans feien inviable una aplicació web ara són possibles gràcies als avenços en aquestes dues especificacions.

Capa de presentació simple i prima

El principi de disseny més important que cal tenir sempre present quan es dissenya la capa web d'una aplicació Java EE és separar la presentació del control del flux i de les crides a la capa de negoci.

La capa de presentació ha de ser el més prima possible. Idealment només hauria de traduir les peticions d'usuari a crides a mètodes de negoci, mai la implementació dels mètodes de negoci.

Ús de *servlets* en la capa de presentació

Hem vist que Java EE recomana l'ús de *Java Server Faces* per a implementar la capa de presentació de les aplicacions web. Amb aquesta premissa us podeu preguntar quan és recomanable utilitzar un *servlet* directament en la capa de presentació.

En general, heu de fer servir *servlets* per a implementar tasques de la capa de presentació que no impliquin la generació de contingut visual. Bàsicament fareu servir *servlets* per al següent:

- Implementar els serveis.

- Fer de controladors en una arquitectura MVC.
- Fer les crides a la lògica de negoci.
- Seleccionar la vista de sortida en una arquitectura MVC.
- Generar contingut binari.
- Eviteu l'ús de *servlets* per a generar contingut textual estàtic.

Tingueu en compte que, moltes vegades, l'ús d'un *framework* per a la capa de presentació involucra un ús "ocult" de *servlets*, i com a desenvolupadors no en fareu servir directament. Per exemple, en l'arquitectura *Java Server Faces* el controlador *FacesServlet* és un *servlet* que us proporciona el *framework* i no heu de desenvolupar.

Crides a la lògica de negoci

Una de les tasques més importants que ha de fer la capa de presentació és cridar la lògica de negoci.

És recomanable crear un component que s'encarregui de centralitzar l'accés a la capa de negoci. Els controladors faran servir aquest component cada cop que hagin de cridar un mètode de negoci.

Aquest component permetrà aïllar la capa de presentació de la implementació que hi hagi en la capa de negoci.

L'ús del component *Business Delegate* és especialment interessant si la lògica de negoci s'implementa amb components distribuïts (per exemple, amb EJB), ja que aquest component s'encarregarà de gestionar la localització dels objectes remots i de tractar les excepcions remotes. Els components de la capa de presentació faran sempre les peticions cap a la capa de negoci mitjançant el *Business Delegate* i aquest s'encarregarà de gestionar tots els aspectes inherents a la implementació de la capa de negoci.

Business Delegate

Aquest disseny segueix el patró de la capa de negoci anomenat *Business Delegate*, que podeu trobar documentat a:

D. Alur; **J. Crupi**; **D. Malks** (2003). *Core J2Ee Patterns: Best Practices and Design Strategies* (2a. Ed.). Upper Saddle River: Prentice Hall.

Tingueu en compte que, moltes vegades, l'ús d'un *framework* per a la capa de presentació ja incorpora una implementació del patró *Business Delegate*.

2.2. Capa de negoci

En aquest subapartat veurem la tecnologia i els components que defineix Java EE per a implementar la capa de negoci.

El model d'aplicacions per a Java EE promou l'ús d'un tipus de components distribuïts anomenats ***EJB***[22] per a implementar la capa de negoci.

[22] De l'anglès *Enterprise Java Beans*.

Veurem quina arquitectura segueixen els components EJB[23], quins serveis ens proporcionen, quins tipus d'EJB defineix l'especificació i per a què serveix cadascun, com s'empaqueten i despleguen, com hi accedeixen els clients i, finalment, les arquitectures més típiques i recomanacions a l'hora d'estructurar la capa de negoci.

[23]La tecnologia d'EJB que analitzarem correspon a la definida en les especificacions 3.x.

La tecnologia d'EJB ha estat la tecnologia més polèmica de totes les que s'inclouen en la plataforma Java EE. També us oferirem una visió crítica, analitzarem els seus punts febles amb una revisió històrica i analitzarem l'especificació d'EJB 3.x, que intenta solucionar aquestes deficiències.

Bibliografia complementària

Podeu trobar els motius que han conduït la polèmica i un bon resum de l'evolució de les especificacions d'EJB en el primer capítol del llibre A. Tate; M. Clark; B. Lee; P. Linskey (2003). *Bitter EJB*. Greenwich: Manning Publications.

2.2.1. Arquitectura EJB

Enterprise Java beans és una tecnologia de components de servidor que permet el desenvolupament i el desplegament d'aplicacions empresarials distribuïdes basades en components.

Nota

Això no vol dir que no es puguin desenvolupar aplicacions escalables, segures, transaccionals, multiusuari i distribuïdes sense fer servir EJB.

Les aplicacions que desenvolupem amb EJB són, almenys teòricament, escalables, segures, transaccionals, multiusuari i distribuïdes. Aquestes aplicacions estan escrites en Java i segueixen un model de components. Això els proporciona un altíssim grau de **reutilització**, i permet que el contenidor implementi els serveis de baix nivell i els desenvolupadors es puguin centrar en la lògica de negoci.

Els **EJB** són components desenvolupats amb Java que compleixen unes especificacions i, per tant, es poden desplegar en qualsevol contenidor d'EJB d'un servidor d'aplicacions compatible amb la plataforma Java EE.

Això podria permetre la creació d'un "mercat" de components EJB de dominis de negoci especialitzats que podríem comprar i assemblar per a construir les aplicacions de negoci en un domini particular.

L'arquitectura dels EJB es basa en els principis següents, que analitzarem al llarg del mòdul:

Bibliografia complementària

Podeu trobar els motius que han conduït la polèmica i un bon resum de l'evolució de les especificacions d'EJB en el primer capítol del llibre A. Tate; M. Clark; B. Lee; P. Linskey (2003). *Bitter EJB*. Greenwich: Manning Publications.

- Els clients només veuen les interfícies que publica l'EJB i hi interactuen cridant els mètodes definits en aquestes interfícies.
- Els EJB s'han de desplegar en un contenidor d'EJB per a poder ser utilitzats pels clients.

- Els clients no criden directament els mètodes de l'EJB. Les crides als mètodes són interceptades pel contenidor abans de passar el control al component.
- El contenidor gestiona les transaccions.
- El contenidor gestiona la seguretat.
- El contenidor tracta tots els temes relacionats amb la concurrència. El desenvolupador programa els EJB com si no tinguéssim un entorn concurrent.
- Les instàncies dels EJB es poden emmagatzemar en bancs d'instàncies per qüestions de rendiment. Això evita haver de crear una instància nova de l'EJB cada cop que un client hi accedeix.
- L'especificació d'EJB defineix una manera estàndard d'empaquetar els EJB i un format per al descriptor de desplegament o les anotacions.

Els EJB com a objectes distribuïts

Els **EJB** es defineixen com a objectes distribuïts, de manera que es poden cridar **remotament**. Els EJB utilitzen RMI com a tecnologia subjacent per a fer comunicacions remotes.

La figura 18 mostra una crida d'un client a un objecte distribuït.

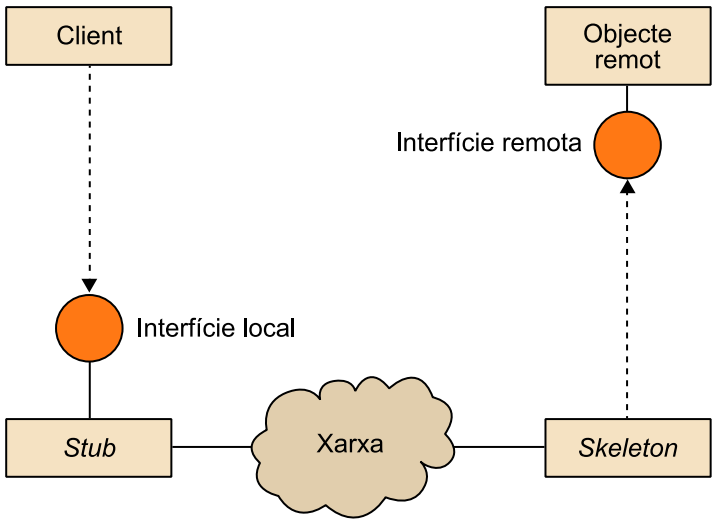

Figura 18. Crida remota

1) El client fa la crida a un objecte que fa de servidor intermediari (*proxy*) a la banda del client (aquest objecte s'anomena ***stub***), que s'encarrega d'aïllar el client de les tasques de baix nivell que implica una crida a un mètode remot. Aquestes tasques inclouen serialitzar la crida (transformar el nom del mètode i els paràmetres a format binari) i enviar-la per la xarxa fins al servidor intermediari de la banda del servidor (***skeleton***).

Vegeu també

En el mòdul "Java RMI" teniu una anàlisi detallada d'aquesta tecnologia i de tots els conceptes que hi estan relacionats (servidors intermediaris –*proxy*–, *stubs*, *skeleton*, serialització, etc.).

2) L'*skeleton* aïlla l'objecte distribuït de les tasques de baix nivell. Quan l'*skeleton* rep una petició de l'*stub*, la desserialitza (reconstrueix la crida remota de format binari a format Java) i fa la crida al mètode de l'objecte remot.

3) L'objecte remot rep la crida de l'*skeleton*, executa el mètode i torna a l'*skeleton* el resultat de l'execució.

4) L'*skeleton* rep el resultat de la crida, el serialitza i l'envia per la xarxa a l'*stub*.

El punt clau per a entendre-ho és que l'objecte remot i l'*stub* tenen la mateixa interfície i els clients tenen la "il·lusió" d'estar cridant directament l'objecte remot.

Els EJB utilitzen RMI com a tecnologia subjacent per a fer comunicacions remotes.

Serveis del contenidor en els EJB

Amb els principis de comunicació distribuïda que hem analitzat en l'apartat anterior solucionem el problema de la distribució d'objectes i som capaços de fer crides a objectes remots com si estiguéssim fent crides locals.

Els EJB segueixen un model de components en el qual el contenidor implementa un conjunt de serveis de baix nivell (seguretat, transaccions, etc.) i els desenvolupadors es poden centrar en la lògica de negoci.

Hi ha dues maneres de fer servir els serveis que proporciona el contenidor: explícitament i implícitament.

Amb l'enfocament **explícit** el contenidor publica l'API per a accedir als serveis i els objectes distribuïts hi accedeixen fent crides a aquesta API.

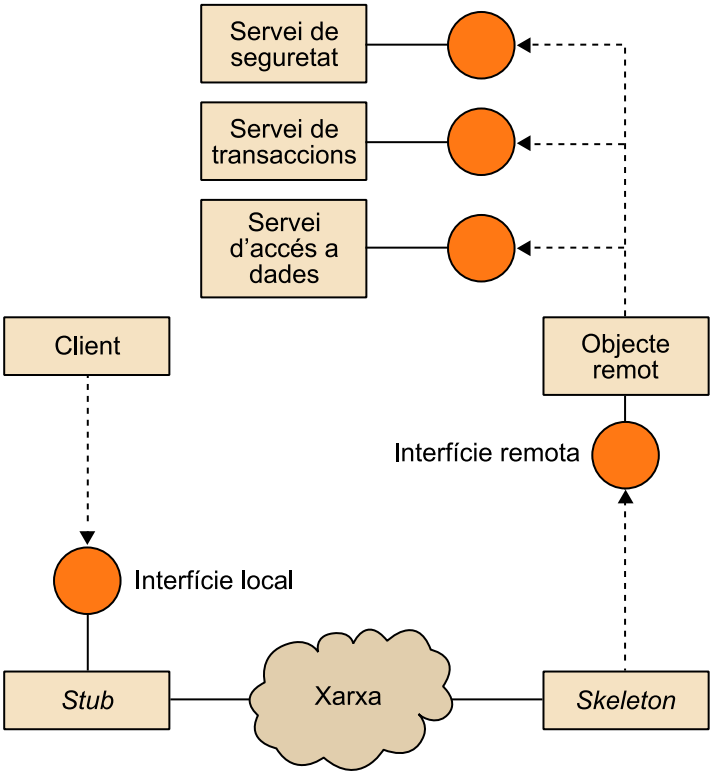

Figura 19. Ús explícit dels serveis

Aquesta aproximació fa que tinguem el codi de negoci barrejat amb el codi que crida els serveis de baix nivell i dóna lloc a desenvolupaments més complexos d'escriure i de mantenir.

Les tecnologies tradicionals d'objectes distribuïts com CORBA, DCOM i RMI fan servir aquesta aproximació.

Vegeu també

Teniu una breu introducció a CORBA i DCOM en el mòdul "Introducció a les plataformes distribuïdes" i una explicació d'RMI en el mòdul "Java RMI" d'aquesta assignatura.

En l'enfocament **implícit** el desenvolupador no crida les API que accedeixen als serveis, sinó que deixa que sigui el contenidor qui faci les crides.

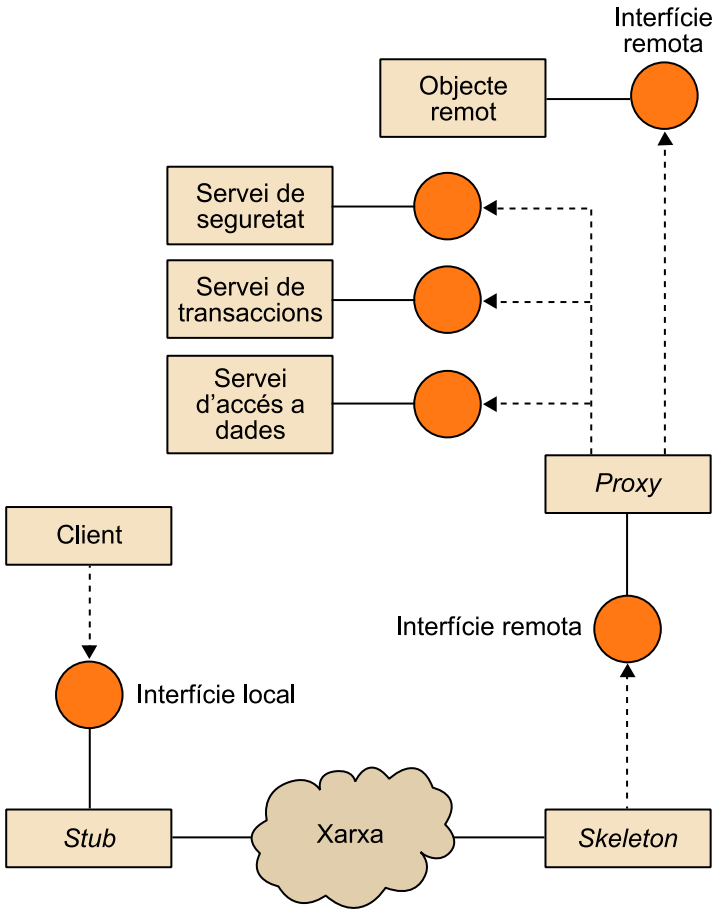

Figura 20. Ús implícit dels serveis

Amb aquesta aproximació, el codi de negoci no està barrejat amb la crida i la gestió dels serveis proporcionats pel contenidor.

- L'objecte distribuït només escriu codi referent a la lògica de negoci.
- El desenvolupador escriu un descriptor de desplegament per a indicar els serveis del contenidor que necessita l'objecte distribuït.
- Hi ha una fase de desplegament de l'objecte distribuït en el contenidor on es genera (habitualment mitjançant una eina proporcionada pel contenidor), a partir del descriptor de desplegament, l'objecte que farà de servidor intermediari entre el client i l'objecte.
- El servidor intermediari intercepta les crides a l'objecte distribuït, executa les crides als serveis associats i passa la crida a l'objecte distribuït per a executar la lògica de negoci.

El *proxy* es genera automàticament a partir de la informació que hi ha en el descriptor de desplegament.

Aquesta aproximació és la que fan servir les noves tecnologies basades en components com els EJB i Microsoft .NET.

Els clients dels EJB tenen la impressió d'estar interactuant directament amb el component EJB desenvolupat. Però, en realitat, totes les peticions passen pel *proxy* abans de ser executades i aquest s'encarrega de fer la feina de baix nivell, cridant i gestionant els serveis d'infraestructura que necessiti el mètode de negoci que el client vol invocar. Els clients mai no criden directament el component EJB, sempre han de passar pel *proxy*.

Servidor intermediari dinàmic

Actualment, els servidors d'aplicacions fan servir *proxys* dinàmics per a fer la intercepció de les crides als components de negoci, i així s'evita la generació en temps de desplegament dels objectes que fan de *proxy* entre el client i el component.

El contenidor proporciona, entre molts altres, els serveis següents de baix nivell:

- **Concurrència**. Gestiona l'accés concurrent als components de manera transparent als clients.

- **Seguretat**. Permet definir declarativament polítiques de seguretat i restriccions d'accés als components, i el contenidor s'assegura de fer-les complir quan un client vol accedir a una funcionalitat exposada per un component.

- **Transaccions**. Permet definir declarativament el comportament transaccional dels mètodes dels components. L'especificació també permet definir el comportament transaccional de manera programàtica, fent servir una aproximació explícita.

- **Disponibilitat i escalabilitat**. Els servidors d'aplicacions han d'assegurar que les aplicacions tinguin alta disponibilitat i siguin escalables.

- **Cicle de vida**. El contenidor s'encarrega de crear i destruir les instàncies dels components segons el tipus de component i el cicle de vida que tingui associat.

- **Persistència**. Permet definir components que gestionin declarativament la persistència de les entitats a bases de dades relacionals. Tal com passa amb les transaccions, també podem definir la persistència programàticament.

2.2.2. EJB 2.x, una visió crítica

Per a entendre el model de programació i desplegament dels EJB 3.x i copsar els avantatges que introdueix, cal fer una revisió del model que es feia servir en les versions anteriors de l'especificació, analitzar-ne els punts febles i veure com els resol EJB 3.x.

En les primeres versions de l'especificació de Java EE semblava que no es podia fer una aplicació Java EE sense fer servir els EJB per a implementar les capes de negoci i d'integració. Fins i tot, des de Sun amb els *blueprints* que contenen les arquitectures de referència es donava una arquitectura basada en EJB com a única alternativa.

Lectura recomanada

El llibre R. Johnson (2004). *Expert One-on-One Java EE Development without EJB*. Indianapolis: Wiley Publishing, mostra algunes alternatives per a desenvolupar aplicacions Java EE amb arquitectures més lleugeres. En aquest llibre també trobareu una llista amb els problemes que pot tenir un mal ús de la tecnologia d'EJB.

Inicialment, es va pensar que s'havien de fer servir EJB per a tot i "estar a l'última" volia dir fer servir EJB en qualsevol desenvolupament empresarial sota la plataforma Java EE.

EJB ***no*** és Java EE. És una tecnologia que podeu escollir si desenvolupeu amb Java EE i que va bé per a resoldre un conjunt de problemes concrets, mentre que pot no ser una bona opció per a altres. No heu de veure els EJB com una solució universal que serveix per a totes les aplicacions.

Els arquitectes i els desenvolupadors triaven els EJB per a implementar aplicacions quan realment no eren la millor opció, i d'això vénen gran part de les crítiques a aquesta tecnologia. Moltes d'aquestes aplicacions van ser un fracàs absolut, i aquests fracàs es va atribuir a l'ús d'EJB per a implementar-les.

Penseu que els EJB es defineixen com a components de negoci distribuïts i, molts cops, es fan servir en arquitectures no distribuïdes, i això provoca problemes de rendiment i complexitat. En aquest cas, no és la tecnologia d'EJB en si el que falla, sinó l'ús inadequat.

Per exemple, una aplicació web que només consulta un catàleg de productes emmagatzemat en una base de dades relacional no requereix molts dels serveis que proporcionen els EJB (seguretat, transaccions, etc.). Fer servir EJB en aquest tipus d'aplicacions és un error i segur que no obtindreu els resultats esperats, i l'increment de complexitat no reportarà cap benefici, al contrari.

Entre les crítiques que va rebre la tecnologia d'EJB en la versió 2.x hi ha les següents:

- **Complexitat**. La tecnologia d'EJB és molt potent, però també força complexa de desenvolupar i de desplegar. Per a crear un component hem de crear diverses classes i interfícies, un descriptor de desplegament complex, sobreescriure mètodes de crida de retorn innecessàriament, etc.

Complexitat d'EJB 2.x

No deixa de ser curiós que la principal crítica que reben els EJB en la versió 2.x sigui la complexitat, precisament quan es van dissenyar com una tecnologia per a ocultar als desenvolupadors la complexitat inherent en una aplicació distribuïda.

- **Rendiment**. Els EJB són objectes distribuïts i en aquests, com a tals, les crides als seus mètodes de negoci són crides remotes. Les crides remotes afegeixen una càrrega important i són la causa de la majoria de problemes de rendiment. Aquest efecte es mitiga amb l'aparició de les interfícies locals.

- **Dificultat per a testejar i depurar**. Les aplicacions que fan servir EJB són força difícils de testejar i depurar, ja que fan servir molts serveis implícits del contenidor.

- **Capa de persistència molt problemàtica**. De tots els tipus d'EJB, els que han rebut més crítiques i han estat menys utilitzats són els EJB d'entitat. Entre les crítiques que reben cal destacar que fins a la versió 2 d'EJB no es podien implementar relacions entre EJB d'entitat; fins a l'aparició de les interfícies locals oferien un rendiment molt pobre; la suposada virtut dels EJB d'entitat és que sigui el contenidor qui gestioni la persistència amb CMP i això tenia greus deficiències fins a la versió 2.1; l'EJB QL era força limitat, etc.

EJB QL

EJB QL és un llenguatge d'especificació de consultes SQL utilitzat per a especificar els mètodes *finder()* i *select()* dels EJB d'entitat en la versió 2.x.

- **Limitacions en l'orientació a objectes**. Els EJB tenen limitacions a l'hora de fer servir característiques bàsiques de l'orientació a objectes, com poden ser l'herència i el polimorfisme.

Finalment, s'ha de tenir en compte que el desenvolupament de components EJB requeria un equip de desenvolupament experimentat.

L'especificació d'EJB 3.x intenta solucionar i mitigar els problemes que hem descrit anteriorment. En el subapartat següent farem una breu introducció a les millores que proposa la nova especificació per tal de resoldre aquests problemes.

2.2.3. EJB 3.x

L'especificació d'**EJB 3.x** se centra, primordialment, a **fer més simple** el desenvolupament i desplegament dels EJB.

Amb les especificacions 2.x d'EJB, desenvolupar un simple EJB que digués "Hola!" implicava crear un mínim de dues interfícies, una classe i un descriptor de desplegament. L'especificació 3.x ho simplifica i resol els punts més criticats de l'especificació:

- Treu la necessitat de definir les interfícies i el descriptor de desplegament. El contenidor les generarà per nosaltres, que les haurem especificat **amb anotacions**. EJB 3.x basa moltes de les seves característiques en les anotacions.

Anotacions

Les **anotacions** són un mecanisme de programació que permet definir metadades en les classes, mètodes, atributs i camps en Java. Les anotacions formen part del JSDK 1.5.

En teniu més informació al JSR 175.

- Es defineixen un conjunt de comportaments per defecte que eliminen la necessitat d'especificar els comportaments més comuns. Si no s'especifica res, el component tindrà el comportament per defecte.

- Simplifica l'accés als EJB des dels clients.

- S'elimina l'obligatorietat d'implementar els mètodes de *callback*.

- Proporciona un conjunt reduït d'API que se centra en el conjunt de casos més comuns i deixa en les especificacions 2.x els escenaris més complexos.

- Facilita el test dels components.

- Ús d'**injecció de dependències** per a facilitar la creació de les instàncies dels objectes.

- Fa servir una aproximació basada en POJO[24] (objectes Java normals) i POJI[25] (interfícies Java normals), i elimina així les restriccions d'herència i polimorfisme que teníem en versions anteriors.

(24)De l'anglès *Plain Old Java Objects*.

(25)De l'anglès *Plain Old Java Interfaces*.

- S'eliminen els EJB d'entitat i es deixa la persistència en mans de les entitats JPA. Amb això el model de persistència guanya molt en simplicitat i s'assembla molt més al model que fan servir els motors de mapatge objecte-relació.

2.2.4. Tipus d'EJB

L'especificació d'EJB defineix dos tipus bàsics d'EJB:

1) EJB de sessió. Modelitzen processos de negoci. En aquest tipus de components hi ha la implementació dels mètodes de negoci.

2) EJB de missatge. Solen modelitzar processos de negoci com els EJB de sessió, però es fan servir per a interactuar amb missatgeria asíncrona.

EJB de sessió

Els **EJB de sessió** són components que serveixen per a implementar la lògica de negoci de les aplicacions Java EE.

Els EJB de sessió no són objectes persistents. El seu temps de vida es limita a una interacció amb el client (que pot ser una única crida o bé un conjunt de crides) i no persisteixen si el servidor d'aplicacions s'atura.

L'especificació d'EJB defineix dos tipus d'EJB de sessió: els EJB de sessió sense estat i els EJB de sessió amb estat.

1) EJB de sessió sense estat

Els **EJB de sessió sense estat** representen processos de negoci que es fan en una única invocació entre el client i el servidor.

Els clients obtenen una referència a l'EJB de sessió sense estat, criden un mètode de negoci i alliberen la referència. Els EJB de sessió sense estat executen una petició síncrona d'un client i li tornen el resultat sense desar-se cap informació que vinculi el client i el component.

Les instàncies no estan lligades a un client. Una mateixa instància d'EJB de sessió sense estat pot servir crides de diferents clients. Les instàncies d'un mateix EJB de sessió sense estat són sempre idèntiques, i el servidor d'aplicacions les sol organitzar en bancs d'instàncies i en va proporcionant als clients perquè aquests executin les crides als mètodes de negoci. Un banc d'instàncies relativament petit pot servir un nombre de clients força elevat.

Bancs d'instàncies

Els **bancs d'instàncies** tenen objectes EJB en memòria llestos per a ser utilitzats pels clients. En fer servir una instància del banc d'instàncies ens estalviem la fase de creació.

Els EJB de sessió sense estat ofereixen un rendiment bastant bo.

Com a exemples d'EJB de sessió sense estat tenim:

- Component que calcula una funció matemàtica.
- Component que rota una imatge 90 graus. El client crida el mètode passant-li la imatge com a paràmetre i el component li torna la imatge rotada 90 graus.
- Component que fa una transferència entre comptes bancaris. El client li proporciona els dos números de compte i l'import i el component fa la transferència.
- Component que verifica el número d'una targeta de crèdit. El client entra el número de la targeta i el component el verifica.
- Component que modelitza un catàleg de productes i ofereix al client la possibilitat de fer cerques i navegar pel catàleg.

Una capacitat important dels EJB de sessió sense estat és que els seus mètodes poden ser publicats com a serveis web en el marc d'una arquitectura SOA. Per a publicar un EJB de sessió sense estat com a servei web només cal afegir-li l'anotació *@WebService* o bé *@WebMethod*.

Vegeu també

En el mòdul "SOA" d'aquesta assignatura s'analitzen amb detall els serveis web i les arquitectures SOA.

2) EJB de sessió amb estat

Els **EJB de sessió amb estat** representen processos de negoci que necessiten diverses invocacions consecutives entre el client i el servidor.

En aquest cas els clients obtenen una referència i la fan servir per a cridar diversos mètodes de negoci abans d'alliberar-la. Els EJB de sessió amb estat emmagatzemen l'estat conversacional amb el client entre diferents peticions. L'estat del client s'emmagatzema en atributs de l'EJB.

Les instàncies estan lligades a un client. Una mateixa instància d'EJB de sessió amb estat no pot servir crides de diferents clients; podem veure cada instància com una sessió amb un client.

Teòricament, en aquest cas el nombre d'instàncies hauria de ser igual al nombre de clients. Això es pot mitigar amb els conceptes de passivació i activació d'instàncies.

La **passivació** consisteix a desar l'estat del component associat a un client al disc i alliberar la instància perquè pugui servir peticions a un altre client. L'**activació** és el procés contrari: recuperar del disc l'estat del client per a continuar servint-li peticions. El client no s'adona d'aquests processos, el servidor d'aplicacions passiva una instància quan fa molt temps que el client no la fa servir i la torna a activar quan en torna a fer una petició.

Vegeu també

En el subapartat "Activació d'objectes remots" del mòdul "Java RMI" d'aquesta assignatura teniu informació detallada dels processos d'activació i passivació.

Els EJB de sessió amb estat ofereixen un rendiment força més dolent que els EJB de sessió sense estat perquè fan un ús molt més gran dels recursos del servidor.

Com a exemples d'EJB de sessió sense estat tenim:

- Component que modelitza la cistella de la compra d'un client en una botiga virtual. El client crea la cistella de la compra i va fent peticions per a afegir i treure objectes de la cistella fins que decideix fer la comanda.
- Component que gestiona un flux de treball. El client arranca un flux de treball que involucra diverses tasques i va fent peticions fins que les completa totes.

EJB de missatge

Els **EJB de missatge** són components sense estat, de servidor, transaccionals que serveixen per a respondre a missatges asíncrons.

Missatge asíncron

En una comunicació asíncrona, l'emissor del missatge (client) no es queda bloquejat esperant la resposta del receptor del missatge, sinó que continua l'execució.

Els EJB de missatge es van introduir en l'especificació 2.0 d'EJB i es poden definir com a consumidors asíncrons de missatges. Des del punt de vista del client, els podem veure com un component que s'executa en el servidor i que fa algun tipus de lògica quan arriba un missatge asíncron.

Tòpics i cues

Els **tòpics** permeten un protocol de publicació/subscripció amb diversos subscriptors per a un mateix missatge, i les **cues**, només un protocol punt a punt amb un únic receptor.

Aquests components es registren (subscriuen) com a consumidors de missatges de cues i tòpics JMS. Tenen un mètode anomenat *onMessage()* que el contenidor crida quan arriba un missatge a algun tòpic o cua al qual estan subscrits. En el mètode *onMessage()* es programa la lògica de negoci associada a aquest esdeveniment. La naturalesa asíncrona dels EJB de missatge fa que no tornin cap resposta al client.

Un EJB de missatge és un EJB complet, com un de sessió, però hi ha diferències importants: té classe EJB i descriptor de desplegament XML, però no exposa interfícies. No li calen, perquè no s'hi accedeix mitjançant l'API Java RMI, només respon missatges asíncrons.

Com a exemples d'EJB de missatge tenim qualsevol procés de naturalesa asíncrona:

- Component que publica una alerta en un panell de control cada vegada que un usuari esgota els tres intents d'accés al sistema.
- Component que emet una factura com a resposta a una comanda.
- Component que s'encarrega de processar els missatges de registre d'una aplicació.

2.2.5. Recomanacions de disseny per a la capa de negoci

Patrons de disseny

Teniu informació detallada dels patrons de disseny més habituals de l'arquitectura Java EE per a la capa de negoci i integració, segons proposa Sun Microsystems, al llibre:

D. Alur; **J. Crupi**; **D. Malks** (2003). *Core J2Ee Patterns: Best Practices and Design Strategies* (2a. Ed.). Upper Saddle River: Prentice Hall.

De la mateixa manera que hem vist que en la capa de presentació calia una fase de disseny per a passar de l'especificació independent de la tecnologia a la implementació amb Java EE, en la capa de negoci també ens cal fer aquesta fase prèvia a la implementació.

En aquesta fase de disseny es prenen decisions d'arquitectura que depenen de cada tecnologia concreta. En aquest subapartat us donarem algunes recomanacions de disseny per a fer el pas de l'especificació a la implementació de la capa de negoci d'una aplicació Java EE. Algunes recomanacions corresponen a l'ús de patrons de disseny i altres us ajudaran a triar entre diferents components per a implementar una funcionalitat determinada.

La primera decisió que s'ha de prendre i que és clau per a la implementació de la capa de negoci és si l'ús d'EJB és la millor alternativa.

Hi ha molts llibres i articles que discuteixen si els EJB són una bona elecció o no. En la bibliografia en teniu alguns que són partidaris de fer servir EJB gairebé sempre, mentre que altres que són partidaris d'utilitzar *frameworks* molt més lleugers.

Spring

L'Spring és un exemple de *framework* lleuger que s'està fent molt popular com a alternativa als EJB.

A l'hora de prendre aquesta decisió cal tenir en compte les importants millores que proporciona EJB 3.x. Alguns autors que no recomanaven l'ús d'EJB 2.x per a la capa de negoci en són més partidaris amb EJB 3.x.

Queda fora de l'abast d'aquests materials estudiar a fons els factors que ens faran decidir si implementem la lògica de negoci amb EJB o no, però, com a norma, podem dir que EJB pot ser una bona alternativa si l'arquitectura final és de naturalesa distribuïda.

Si, finalment, decidiu que necessiteu utilitzar EJB per a l'arquitectura de l'aplicació que esteu desenvolupant, cal que prengueu una sèrie de decisions sobre **com** utilitzar-los o **quin** tipus és més adient.

Accés local o remot

Els EJB són components de negoci distribuïts i, per tant, han de proporcionar accés remot als clients. L'accés remot a un component té un conjunt de processos associats que provoquen un *overhead* important a cada crida.

En moltes arquitectures, els clients dels EJB i els components amb els quals interactuen es troben a la mateixa màquina virtual, i això fa innecessari un accés remot.

A partir de l'especificació 2.x d'EJB, es poden definir accessos tant locals com remots als EJB, i es permet que les crides que no hagin de ser remotes es comportin com a locals i no pateixin tant *overhead*. En aquest sentit, pren molta força el patró façana que analitzarem en l'apartat següent.

En el mòdul "Desenvolupament de programari basat en components" hem vist com cal definir la distribució física dels components lògics de l'aplicació (especificada mitjançant diagrames de desplegament). D'acord amb aquesta distribució, cal que destrieu quins components seran accedits remotament i, per a aquests, habilitar l'accés remot.

El criteri per a triar l'accés als components EJB com a remot és si aquests han de ser accedits des de fora de la màquina virtual en què s'executen, en qualsevol altre cas cal triar un accés local.

Accés remot

El rendiment d'un component accedit remotament pot arribar ser cent cops més lent que el mateix component accedit localment.

Moltes vegades no té sentit proporcionar un accés local i un altre de remot per al mateix component perquè el fet que la crida sigui remota o local influeix molt en la granularitat dels mètodes que exposi el component per a cada tipus d'interfície.

L'elecció del tipus d'accés a un component influeix en la manera com s'ha de dissenyar.

Bibliografia complementària

Teniu una extensa explicació d'aquesta qüestió al llibre R. Johnson (2003). *Expert One-on-One Java EE Design and Development* (pàg. 224). Indianapolis: Wiley Publishing.

Patró façana aplicat als EJB

El **patró façana** proporciona un punt d'entrada unificat per a un conjunt de serveis de la capa de negoci de l'aplicació.

L'aplicació d'aquest patró a les aplicacions Java EE es redueix a definir un EJB de sessió que encapsuli l'accés als objectes de negoci de l'aplicació. Els clients només interactuaran amb aquest component per a fer peticions a la capa de negoci (vegeu les figures 21 i 22).

Session facade

Teniu tota la informació referent al patró de disseny *session facade* al llibre:

D. Alur; **J. Crupi**; **D. Malks** (2003). *Core J2Ee Patterns: Best Practices and Design Strategies* (2a. Ed.). Upper Saddle River: Prentice Hall.

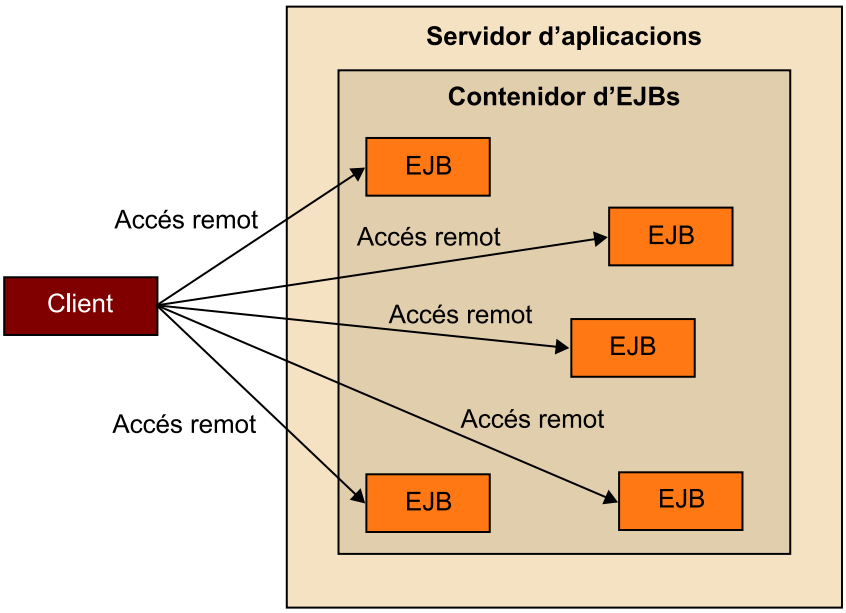

Figura 21. Accés sense patró façana

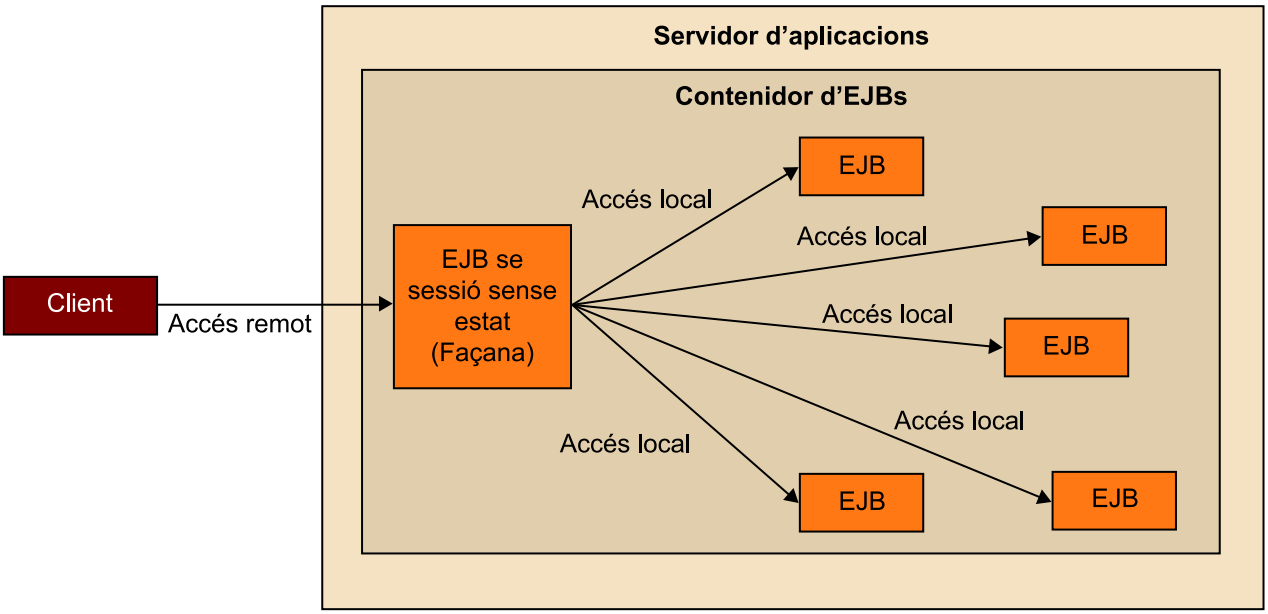

Figura 22. Accés amb patró façana

El patró façana és especialment important si feu servir accessos remots als components de negoci. Aquest patró permet fer només crides remotes a l'EJB que fa de façana i que les crides d'aquest als components de negoci siguin locals, de manera que es redueix així l'*overhead* inherent a les comunicacions remotes.

L'ús del patró de disseny façana per a accedir als mètodes de negoci proporciona els avantatges següents:

- Dóna lloc a un sistema més cohesionat i menys acoblat.
- Proporciona als clients una interfície **simplificada** i **unificada** d'accés a la lògica de negoci de l'aplicació.
- Permet aïllar als clients de canvis en els objectes de negoci.
- Redueix el nombre d'objectes de negoci que s'exposen als clients, i es disminueix així l'acoblament entre capes.
- Oculta als clients les interdependències que hi pugui haver entre els components de la capa de negoci.

2.3. Capa d'integració

En aquest subapartat veurem la tecnologia i els components que defineix Java EE per a implementar la capa d'integració.

El model d'aplicacions per a Java EE promou l'ús de JPA[26] per a implementar la capa d'integració amb bases de dades relacionals.

[26] De l'anglès *Java Persistence API*.

JPA ofereix un *framework* de mapatge objecte-relació entre objectes Java i taules d'una base de dades relacional, i presenta els caràcters següents:

- Simplifica el desenvolupament d'aplicacions que hagin d'accedir a dades emmagatzemades en bases de dades relacionals i manipular-les.
- Ofereix persistència d'objectes de manera transparent.
- Fa que el nostre codi no sigui depenent d'un sistema gestor de base de dades concret.
- Defineix com s'ha de descriure el mapatge entre objectes i taules.
- Defineix la interfície amb el proveïdor de persistència; aquest proveïdor serà qui interactuï amb la base de dades.

L'especificació de JPA defineix el següent:

- Com s'ha de descriure el mapatge entre objectes i taules mitjançant fitxers XML o *anotacions*.
- Especifica la interfície *EntityManager*, que gestiona el cicle de vida de les entitats JPA i les operacions de persistència que es poden fer amb les entitats.
- Especifica el llenguatge de consulta JPQL[27], llenguatge semblant a SQL però que treballa amb entitats en lloc de fer-ho amb taules.

Recordeu que si volem accedir a fonts de dades que no siguin bases de dades relacionals haurem de fer servir JCA[28] o bé JDO[29]. Aquests tipus d'accés queden fora de l'abast dels materials.

[27] De l'anglès *Java Persistence Query Language*.

[28] De l'anglès *Java Connector Architecture*.

[29] De l'anglès *Java Data Objects*.

2.3.1. Entitats JPA

A partir de la versió 5 de Java EE, el suport de la plataforma a la persistència d'entitats en bases de dades relacionals es fa mitjançant entitats JPA, que substitueixen els polèmics EJB d'entitat.

Les **entitats JPA** són components que representen les dades persistents del model de negoci de l'aplicació. Els podeu veure com una representació Java, en memòria i en forma d'objecte, de la informació emmagatzemada en una base de dades relacional.

Les entitats JPA **NO** són exclusives de la plataforma Java EE ni de la capa d'integració, i això vol dir que es poden utilitzar entitats JPA en aplicacions JSE[30].

[30]De l'anglès *Java Standard Edition*.

JSE

Les aplicacions estàndard de Java es desenvolupen amb JSE (*Java Standard Edition*).

Les entitats JPA són objectes **persistents** perquè duren el mateix que les dades que representen, duren més enllà que el temps de vida dels components i, fins i tot, i a diferència del EJB de sessió, més enllà que el temps de vida del servidor d'aplicacions. Tot i que, tècnicament, els components no duren mai més enllà del temps de vida del servidor d'aplicacions, les dades que representen les entitats JPA sí que continuen existint encara que el servidor d'aplicacions caigui, i per això s'acostuma a dir que les entitats JPA duren més enllà que el temps de vida del servidor d'aplicacions.

Els EJB d'entitat són objectes intel·ligents en el sentit que saben com gestionar la seva persistència de manera més o menys transparent al desenvolupador. Les entitats JPA mapen (estableixen una correspondència) els seus atributs en camps de taules de les bases de dades i són capaços d'actualitzar-les quan es modifiquen els atributs.

Normalment les entitats JPA **no** tenen lògica de negoci, tret de manipulacions simples de les dades que representen.

Una entitat JPA representa una taula en una base de dades relacional, una instància d'una entitat JPA representa una fila de la taula i cada camp persistent de l'entitat representa una columna de la taula.

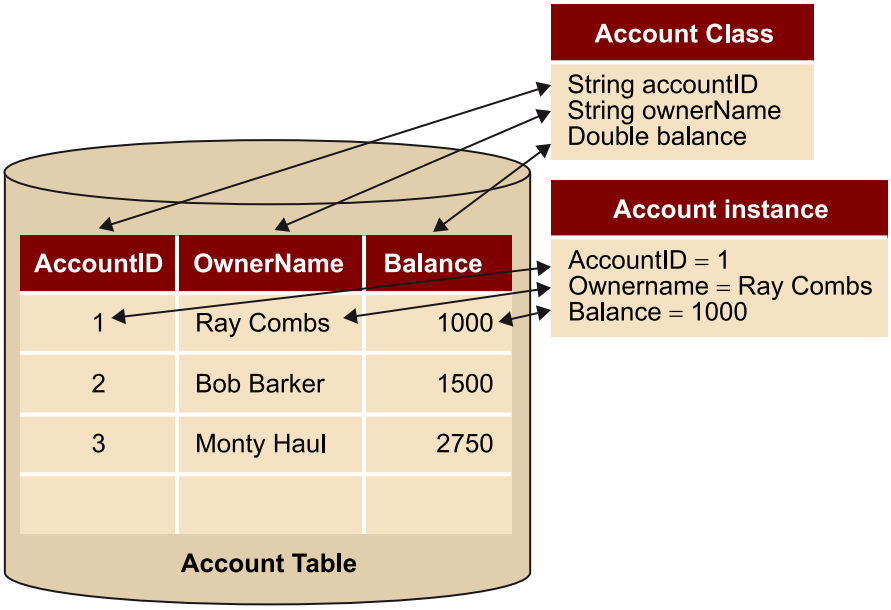

Figura 23. Exemple de mapatge entre una entitat JPA i una taula de la base de dades

Per a especificar una entitat JPA cal tenir en compte els aspectes següents:

- Cada entitat JPA és una classe Java que compleix amb les restriccions dels *Java beans*, marcada amb l'anotació *@Entity*, i correspon a una taula de la base de dades.

- Cada instància de la classe Java es mapa amb una fila de la taula, i això implica que les instancies de les entitats han de ser úniques i han d'estar identificades per un o diversos dels seus atributs.

- Tots els atributs de l'entitat, excepte els decorats amb l'anotació *@Transient*, són persistents i es maparan amb columnes de la taula.

Java beans

Recordeu que els *Java beans* són classes Java normals que han de tenir un constructor buit i accés amb *getXXX* i *setXXX* a les seves propietats.

Les entitats JPA permeten especificar les relacions més habituals entre les taules d'una base de dades relacional:

- *@OneToOne*: especifica una relació 1:1.
- *@OneToMany*: especifica una relació 1:N, s'aplica a atributs de tipus *Set* o *Collection*.
- *@ManyToOne*: especifica una relació N:1.
- *@ManyToMany*: especifica una relació N:M, s'aplica a atributs de tipus *Set* o *Collection*.
- *@JoinTable*: especifica que una relació *ManyToMany* o *ManyToOne* fa ús d'una taula auxiliar.

Anotacions

En l'especificació de JPA teniu totes les anotacions que fan possible l'especificació completa del mapatge entre una entitat JPA i la taula corresponent de la base de dades, com per exemple, definició dels atributs identificadors, mapatge entre noms d'atribut i noms de camps, camps nuls, etc.

Com a exemples d'entitats JPA tenim:

- Component que representa la informació d'un client, amb el nom, el DNI i l'adreça.
- Component que representa la informació dels productes d'un catàleg; per exemple, amb l'identificador del producte, el nom, una imatge i el preu.
- Component que representa una notícia penjada en un fòrum de discussió.

Per exemple, si volem modelitzar l'esquema de taules que presenta la figura 24 amb entitats JPA:

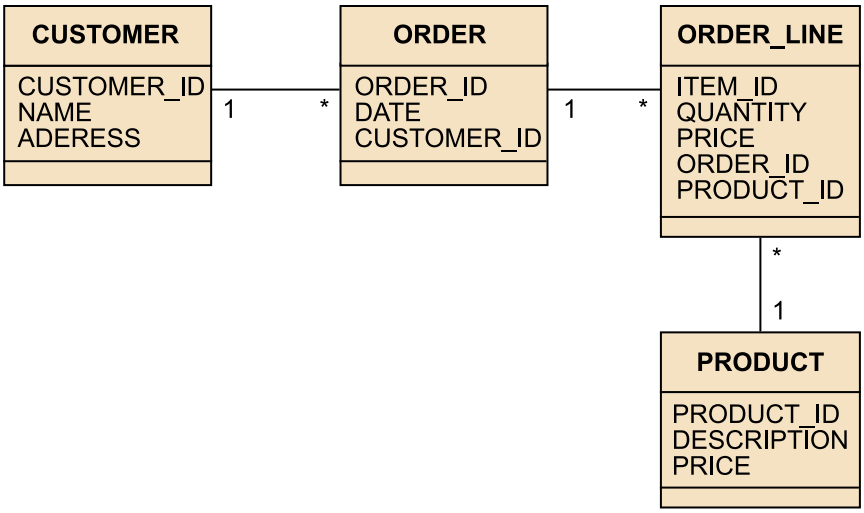

Figura 24. Exemple de taules per a fer el mapatge a entitats JPA

Ho podríem fer així:

```
@Entity @Table(name = "PRODUCT")
public class Product implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "PRODUCT_ID") private Integer productId;
    @Column(name = "DESCRIPTION") private String description;
    @Column(name = "PRICE") private float price;
    ...
}
@Entity @Table(name = "ORDER")
public class Order implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "ORDER_ID")
    private Integer orderId;
    @Column(name = "DATE") @Temporal(TemporalType.DATE)
    private Date orderDate;
    @JoinColumn(name = "CUSTOMER_ID", referencedColumnName = "CUSTOMER_ID")
    @ManyToOne
    private Customer customer;
    @OneToMany(mappedBy="order", cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    private Collection<OrderLine> orderLines;
    ...
}
@Entity @Table(name = "CUSTOMER")
public class Customer implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "CUSTOMER_ID") private Integer customerId;
    @Column(name = "NAME") private String name;
    @Column(name = "ADDRESS") private String address;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "customer")
    private Collection<Order> orders;
    ...
}
@Entity @Table(name = "ORDER_LINE")
public class OrderLine implements Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "ITEM_ID") private Integer itemId;
    @Column(name = "QUANTITY") private float quantity;
    @Column(name = "PRICE") private float price;
    @JoinColumn(name = "PRODUCT_ID", referencedColumnName = "PRODUCT_ID")
    @ManyToOne
    private Product product;
    @JoinColumn(name = "ORDER_ID", referencedColumnName = "ORDER_ID")
    @ManyToOne
    private Order order;
    ...
}
```

Figura 25. Codi de les entitats JPA

2.3.2. La interfície *EntityManager*

La interfície *EntityManager* ofereix mètodes per a poder interactuar amb l'entorn de persistència: control del cicle de vida de les entitats, gestió de la memòria cau, control de transaccions, creació i execució de consultes, etc.

Podríem dir que *EntityManager* connecta les entitats amb la base de dades concreta. El primer que cal fer per a interactuar amb les entitats JPA és obtenir una referència a l'*EntityManager* que les gestionarà. La referència a l'*EntityManager* es pot obtenir amb injecció de dependències o mitjançant la factoria *EntityManagerFactory*.

Totes les operacions que es facin amb l'*EntityManager* s'han de fer dins un context transaccional; si es criden des d'un EJB o des de components web les transaccions seran gestionades pel servidor d'aplicacions, i si no tenim context transaccional l'*EntityManager* mateix ofereix operacions per a treballar amb transaccions.

L'*EntityManager* ofereix un conjunt de mètodes per a treballar amb el cicle de vida de les entitats: crear una instancia d'una entitat, fer persistents els canvis a una instància d'una entitat, eliminar una instància d'una entitat, actualitzar els valors dels atributs d'una instància d'una entitat amb el contingut de la fila de la base de dades que la representa, sincronitzar totes les instàncies que està gestionant, cercar una instància a partir de la seva clau primària, etc.

Especificació de JPA

En l'especificació de JPA teniu tots els mètodes que ofereix la interfície *EntityManager*.

2.3.3. Llenguatge de consulta JPQL

L'especificació de JPA defineix un llenguatge molt similar a SQL, anomenat *JPQL*[31], per a fer consultes de tipus *select, update* i *delete* a les entitats JPA.

(31)De l'anglès *Java Persistence Query Language*.

JPQL és un llenguatge estàndard definit en l'especificació de JPA, que serveix per a fer consultes a les entitats JPA.

JPQL és molt similar a SQL, però treballa sobre entitats JPA, en lloc de fer-ho directament amb les taules de la base de dades subjacent. Això permet fer codi portable entre diferents sistemes gestors de bases de dades.

2.3.4. Recomanacions de disseny per a la capa d'integració

La capa d'integració és una de les capes que presenta més opcions en les aplicacions Java EE. Una aplicació Java EE pot implementar la capa d'integració amb el següent:

- POJO
- *Frameworks* de mapatge objecte-relació de tercers
- JPA

La implementació de la persistència de les aplicacions Java EE amb **POJO** és, possiblement, l'opció més senzilla. Es defineixen els objectes persistents que modelitzen les entitats com a classes Java normals i, en l'opció més simple, es fa servir JDBC escrit pel programador per a mantenir-les.

Si l'aplicació que desenvolupem té un model de dades simple, basat en accessos bàsicament de lectura, i no requereix un comportament transaccional elevat, ni accés a fonts de dades heterogènies, l'ús de POJO pot esdevenir una bona elecció.

Els *frameworks* de mapatge objecte-relació de tercers són una de les alternatives més usades per a implementar la persistència en la capa d'integració de les aplicacions Java EE. Són sistemes que automatitzen el procés de creació, lectura, actualització i escriptura d'entitat en les bases de dades relacionals i, la majoria, ofereixen una eina completa de mapatge objecte-relació. Molts cops les eines de mapatge objecte-relació basen la seva arquitectura en POJO.

Exemples de *frameworks*

Exemples de *frameworks* de mapatge objecte-relació: Hibernate, Castor, Jakarta OJB, TopLink.

Els avantatges més importants d'aquestes eines són la facilitat d'ús, la persistència transparent d'entitats, moltes optimitzacions de rendiment, generació automàtica de les classes Java que representen les entitats i les relacions a partir de fitxers de definició, etc. El principal desavantatge és que **no** és una tecnologia estàndard.

Les **entitats JPA** són el mecanisme de persistència que impulsa l'especificació de Java. Com hem vist, es tracta d'un *framework* de mapatge objecte-relació que proporciona els avantatges que ofereixen les eines de mapatge objecte-relació i soluciona el seu principal desavantatge: es tracta d'una tecnologia estàndard.

Hibernate

L'especificació de JPA està basada en l'experiència obtinguda en *frameworks* de mapatge objecte-relació de tercers com Hibernate.

Les entitats JPA apareixen en l'especificació de Java EE com a resposta al fracàs dels EJB d'entitat, els components de persistència que preveia Java EE fins a la versió 5.

2.4. Capa de serveis transversals

La **capa de serveis transversals** proporciona accés als components de qualsevol capa a serveis necessaris. La característica principal d'un servei transversal és el fet de ser accessible des de qualsevol capa de l'aplicació. Aquests serveis poden ser molts i molt variats; entre els més comuns, podem trobar serveis de seguretat (autenticació, autorització i validació), de registre, de monitoratge, d'injecció de dependències, de configuració, de gestió de la memòria cau i molts més.

Nota

Queda fora de l'àmbit d'aquests materials fer una llista exhaustiva del suport que ofereix JEE per a implementar els serveis transversals que pugui necessitar una aplicació.

El servei de seguretat és un bon exemple de servei transversal, cal mantenir els contextos de seguretat entre totes les capes de l'aplicació i fer que la gestió de la seguretat global en tota l'aplicació sigui coherent.

El servei de registre (*logging*) és un altre exemple de servei transversal: qualsevol component ha de poder accedir al servei de registre per a registrar informació en el sistema, independentment de la capa a la qual pertanyi.

2.5. Arquitectures habituals per a aplicacions Java EE

Aplicacions Java EE

En la primera edició del llibre de referència de patrons Java EE i en els *blueprints* semblava que només es podia implementar la lògica de negoci d'una aplicació Java EE fent servir EJB.

Molts cops sembla que crear una aplicació sota una arquitectura Java EE és molt complex i que qualsevol desenvolupament, per petit i simple que sigui, es torna una tasca feixuga.

Aquesta sensació és deguda al fet que, moltes vegades, si seguim l'arquitectura tradicional recomanada per Sun, obtenim una arquitectura **sobredimensionada** i molt complexa que no ens cal per al problema que intentem resoldre. Inicialment semblava que Sun proposava **sempre** una arquitectura complexa.

Lectura recomanada

El llibre R. Johnson (2004). *Expert One-on-One Java EE Development without EJB*. Indianapolis: Wiley Publishing, mostra algunes alternatives per a desenvolupar aplicacions Java EE amb arquitectures més lleugeres.

El sobredimensionament de l'arquitectura de les aplicacions Java EE porta, sovint, a aplicacions complexes i molt difícils de mantenir.

Usualment, es té la sensació que Java EE = EJB, i això **no** és cert. Es poden desenvolupar aplicacions Java EE amb arquitectures que tinguin o no EJB.

En aquest subapartat analitzarem, a tall d'exemple, diferents arquitectures que podeu definir sota la perspectiva tecnològica de Java EE d'acord amb la distribució dels components que formen l'aplicació.

La **millor** arquitectura per a un sistema és la **més simple** que compleixi amb els requisits.

2.5.1. Arquitectures amb components no distribuïts

Bibliografia complementària

Consulteu els avantatges i desavantatges de cadascuna de les arquitectures exposades al llibre R. Johnson (2003). *Expert One-on-One Java EE Design and Development*. Indianapolis: Wiley Publishing.

Són arquitectures en què els components s'executen en una mateixa màquina virtual, no tenim una aplicació realment distribuïda i tots els components s'executen al mateix servidor físic.

Es pot considerar una aplicació no distribuïda quan la capa client i la capa EIS estan en màquines separades, però les capes de presentació, negoci i integració i tots els components que les formen es troben desplegats a la mateixa màquina, de fet, al mateix servidor d'aplicacions.

Aplicació web sense fer servir EJB per a la capa de negoci

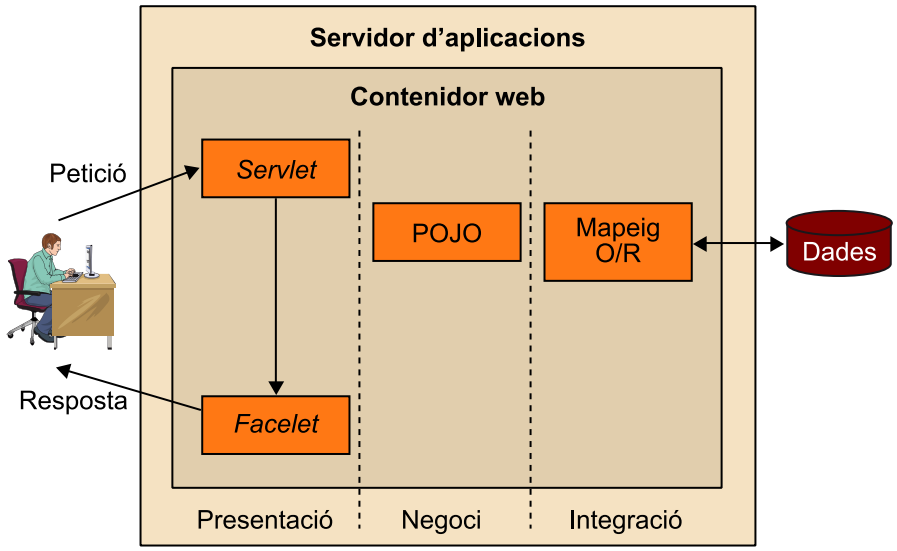

Figura 26. Arquitectura web sense EJB

Aquesta és l'arquitectura més simple. Els components web i els components de negoci s'executen en la mateixa màquina virtual i els components de negoci **no** són EJB, i només es necessita un contenidor web. Amb tot, hi ha una clara separació entre els components de negoci i els components de presentació.

En aquesta arquitectura tenim:

- Capa de presentació amb *Java Server Faces* amb un patró de disseny MVC.
- Accés local de la capa de presentació a la capa de negoci.
- Capa de negoci implementada amb POJO, possiblement amb *Managed Beans*.
- Capa d'integració normalment desenvolupada amb algun *framework* de mapatge objecte-relació (entitats JPA o altres *frameworks* de tercers), o bé directament amb classes Java seguint el patró DAO[32].

[32] De l'anglès *Data Access Object*.

Vegeu també

Teniu més informació del patró de disseny DAO en l'assignatura *Anàlisi i disseny amb patrons*.

Aplicació web amb EJB locals per a la capa de negoci

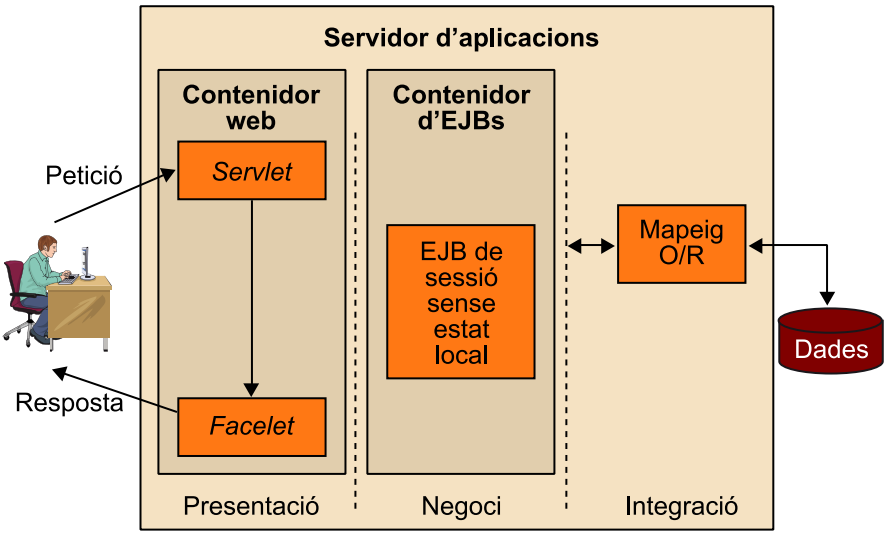

Figura 27. Arquitectura web amb accessos locals

El contenidor web i el contenidor d'EJB i la capa d'integració s'executen en la mateixa màquina virtual i l'accés entre totes les capes és **local**. Aquesta arquitectura és similar a l'anterior, però ara els components de negoci són EJB, als quals s'accedeix localment des dels components de presentació, en lloc de ser POJO. Els components de negoci i els components web s'executen en la mateixa màquina virtual.

En aquesta arquitectura tenim:

- Capa de presentació amb *Java Server Faces* amb un patró de disseny MVC.
- Accés local de la capa de presentació a la capa de negoci.
- Façana d'accés a la capa de negoci amb EJB de sessió sense estat locals.
- Capa d'integració, normalment desenvolupada amb algun *framework* de mapatge objecte-relació (entitats JPA o altres *frameworks* de tercers), o bé directament amb classes Java segons el patró DAO[33].

[33] De l'anglès *Data Access Object*.

2.5.2. Arquitectures amb components distribuïts

És una arquitectura en què els components de negoci poden estar distribuïts en diverses màquines (o a la mateixa màquina física, però executant-se en màquines virtuals diferents) i, per tant, les crides a aquests han de ser remotes.

Tenim dues opcions: fer servir EJB per a la capa de negoci o bé serveis web.

Arquitectura distribuïda amb EJB

És una arquitectura força més complexa que permet als clients fer crides remotes als mètodes de negoci. La comunicació entre la capa de presentació i la capa de negoci és distribuïda i, fins i tot, la comunicació entre diferents components de la capa de negoci pot ser distribuïda. El contenidor web i el contenidor d'EJB s'executen en màquines virtuals diferents i l'accés entre aquestes dues capes és **remot**. L'accés entre l'EJB de sessió i les entitats JPA és **local**. És possible tenir clients que accedeixin a l'EJB directament de manera remota.

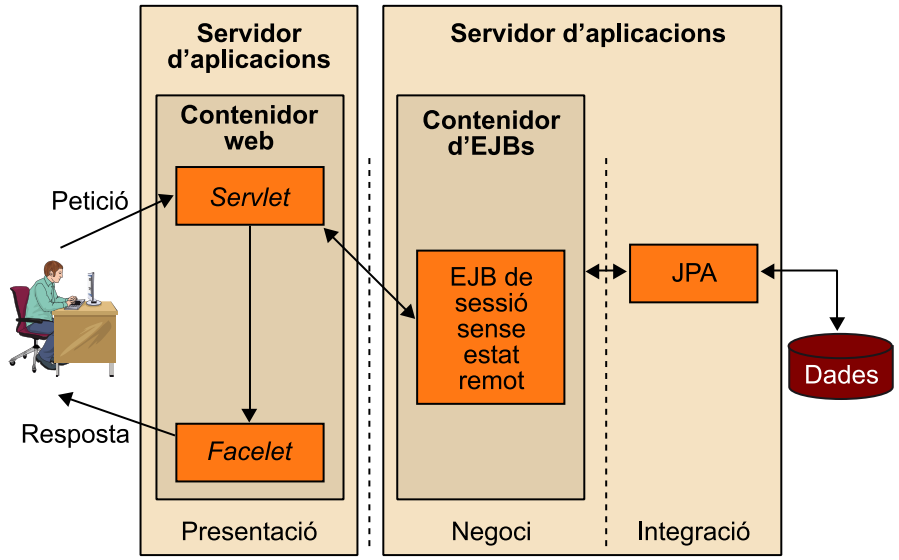

Figura 28. Arquitectura distribuïda amb EJB

En aquesta arquitectura, que és la que proposa Sun en els *blueprints*, tenim:

- Capa de presentació amb *Java Server Faces* segons un patró de disseny MVC.
- Accés remot de la capa de presentació a la capa de negoci mitjançant RMI sobre IIOP.
- Façana d'accés a la capa de negoci amb EJB de sessió sense estat remots.
- Capa d'integració amb entitats JPA i accedits localment des dels EJB de sessió que fan de façana.

En aquesta arquitectura tenim components de negoci distribuïts. En molts casos aquesta arquitectura és més robusta i escalable, però també és força més complexa i pot arribar a proporcionar un rendiment pitjor.

Arquitectura distribuïda amb serveis web

Arquitectura molt similar a l'anterior, però que permet a la capa de negoci interactuar amb clients que no suportin la tecnologia Java EE, mitjançant serveis web, i permet publicar serveis de la nostra capa de negoci com a serveis web perquè puguin ser cridats per altres components externs i heterogenis.

3. Disseny d'aplicacions Java EE amb UML

En mòduls anteriors hem vist com utilitzar RM-ODP per a definir l'arquitectura en termes de components arquitectònics i connectors entre aquests i com, posteriorment, es pot refinar aquesta especificació en termes d'un conjunt de components de programari que els implementen. No obstant això, els components de programari estan especificats de manera independent a la tecnologia que utilitzarem per a codificar l'aplicació.

Ja hem vist en els subapartats "Recomanacions de disseny per a la capa de presentació", "Recomanacions de disseny per a la capa de negoci" i "Recomanacions de disseny per a la capa d'integració" que per a passar de l'especificació independent de la tecnologia a la implementació amb una tecnologia de components determinada (en aquest cas Java EE), cal fer una fase prèvia de disseny per tal d'evitar errors i reduir la distància existent entre l'especificació i la implementació concreta. Aquesta fase de disseny és especialment important amb vista a aconseguir que els programadors de l'aplicació tinguin tota la informació necessària per a poder implementar l'aplicació tal com nosaltres, els dissenyadors, volem que es faci. Això implica, òbviament, que hem de ser capaços de representar dins els diagrames UML aquestes decisions.

Vegeu també

En els mòduls "Arquitectura del programari" i "Desenvolupament de programari basat en components" d'aquesta assignatura s'aconseguia una especificació independent de la tecnologia.

Aquesta fase de disseny partirà, doncs, d'una representació dels components que formen l'aplicació independent de la tecnologia i finalitzarà amb una representació d'aquests mateixos components amb Java EE. Aquesta representació amb Java EE inclourà les decisions de disseny necessàries per a permetre una implementació fàcil de l'aplicació dissenyada.

Vista tecnològica

Teniu en compte que, en el cas que ens ocupa, la tecnologia concreta serà Java EE, però podríem fer els mateixos raonaments si la tecnologia concreta que representem amb la vista tecnològica fos una altra, com per exemple .NET.

Dins de l'ampli conjunt de diagrames que especifica UML 2.x, ens centrarem en com es representa el sistema concret amb els diagrames de components i de desplegament. Anteriorment, en aquests materials, heu vist un resum molt breu d'UML i els mecanismes d'extensió que proporciona, amb especial atenció als perfils.

Vegeu també

Vegeu l'assignatura *Enginyeria del programari* per a una visió més completa d'UML i consulteu el mòdul "Desenvolupament de programari basat en components" d'aquesta assignatura, per a veure els mecanismes d'extensió que proporciona UML.

En aquest apartat analitzarem com es fa la fase de disseny per a passar de l'especificació a la implementació; aprendrem a representar la vista tecnològica d'un sistema fent servir UML i, més concretament, un perfil UML per a Java EE; i veurem com s'especifiquen els diagrames de desplegament i de components per a la plataforma Java EE. Per a fer-ho, ens basarem en un exemple senzill que il·lustri els passos que cal fer i algunes de les decisions que cal considerar.

3.1. Un perfil UML per a aplicacions Java EE

UML és un llenguatge de modelització estàndard adient per a representar els components de programari que formaran l'aplicació. Com que UML és un llenguatge de propòsit general i, com a tal, es pot quedar curt per a modelitzar aspectes específics d'un determinat domini o tecnologia, necessitem mecanismes d'extensió de la sintaxi i la semàntica d'UML per a aconseguir modelitzar els aspectes específics d'un domini o tecnologia particulars.

Els **perfils d'UML** són el mecanisme que actualment proporciona UML per a estendre'n la sintaxi i la semàntica i poder modelitzar les particularitats d'un domini o tecnologia. Aquests perfils són especialment importants per a definir amb exactitud la vista tecnològica del sistema i ens permetran modelitzar les particularitats de cada tecnologia concreta amb un llenguatge de propòsit general.

En el cas que ens ocupa hem escollit Java EE com a tecnologia d'implementació, que presenta algunes particularitats que fan necessària l'extensió d'UML per a poder representar-les.

Algunes variacions o particularitats de Java EE respecte als conceptes generals que defineix UML i que heu pogut veure anteriorment són:

- Depenent del tipus de component (EJB de sessió, EJB de missatge, entitat JPA, etc.) els mètodes per defecte que cal implementar són diferents.
- Els EJB poden tenir dos tipus d'accés: remots i locals. Hem de poder representar quin volem fer servir en cada cas.
- Cal representar els descriptors de desplegament i les anotacions associades als components.
- Cal especificar els formats estàndard d'empaquetament per a aplicacions Java EE, components web i components EJB.

Si no especialitzéssim UML per a la representació d'aplicacions Java EE, caldria definir tots aquests detalls per mitjà de notes dins el diagrama, cosa que el faria molt complicat de llegir i tractar.

Cal, doncs, utilitzar els mecanismes d'extensió per a reflectir les particularitats dels artefactes que formen una aplicació Java EE: cal definir un **perfil UML** per a **Java EE**.

Per exemple, el perfil UML per a EJB defineix un **component estereotipat** `<<EJBSessionBean>>` amb tota la semàntica que defineix els components EJB de sessió que hem analitzat anteriorment.

Inicialment, es va fer una proposta de perfils d'EJB emmarcat dins el perfil EDOC de l'OMG, però es va retirar l'any 2004. Més informació a http://www.jcp.org/en/jsr/detail?id=26.

A falta d'una definició estàndard per al perfil de Java EE, per a redactar aquests materials ens basarem en el perfil de Java EE/EJB que apareix com a exemple al document de *Superestructura d'UML 2.x* i hi afegirem els que necessitem per tal de modelitzar els aspectes més rellevants d'una aplicació Java EE. La taula següent mostra alguns dels estereotips que defineix el document de *Superestructura d'UML 2.x* i altres que hem definit en aquests apunts per tal d'il·lustrar conceptes que hem vist en els materials, però no apareixen en l'exemple d'aquest perfil.

Web recomanat

Teniu el document de *Superestructura d'UML 2.x* a http://www.omg.org/spec/UML.

En aquest document hi ha una definició de perfil per a Java EE/EJB molt simple. No és una especificació formal, però ens servirà per a documentar l'ús i la definició de perfils dins l'especificació 2.x d'UML.

Estereotip	Tipus d'element	Descripció
<<EJBEnterpriseBean>>	Component	El component representa un EJB.
<<EJBSessionBean>>	Component	El component representa un EJB de sessió.
<<EJBMessageDrivenBean>>	Component	El component representa un EJB de missatge.
<<JPAEntity>>	Component	El component representa una entitat JPA.
<<EJBRemoteInterface>>	Interfície	Interfície remota d'un component EJB.
<<EJBLocalInterface>>	Interfície	Interfície local d'un component EJB.
<<EJBLocaMethod>>	Mètode	Mètode de negoci amb accés local d'un component EJB.
<<EJBRemoteMethod>>	Mètode	Mètode de negoci amb accés remot d'un component EJB.
<<EJBRoleName>>	Actor	Nom del rol de seguretat que serveix per a definir permisos.
<<JavaSourceFile>>	Artefacte	Fitxer font Java.
<<JAR>>	Artefacte	Fitxer en format JAR (*Java Archive*). Conté un conjunt de classes Java compilades i fitxers de recursos.
<<EJBDeployment>>	Artefacte	Fitxer en format JAR que representa un component EJB o diversos components EJB empaquetats.
<<JSF>>	Component	*framework Java Server Faces* per a la capa de presentació.
<<JSP>>	Component web	El component representa una JSP.
<<facelet>>	Component web	El component representa un *facelet*.
<<ManagedBean>>	Component web	El component representa un *Java Bean* gestionat pel *framework Java Server Faces*.
<<action>>	Mètode	Mètode d'acció dins el marc *Java Server Faces*.
<<servlet>>	Component web	El component representa un *servlet*.
<<ear>>	Artefacte	Fitxer en format EAR (*Enterprise Archive*). Representa una aplicació Java EE empaquetada.
<<war>>	Artefacte	Fitxer en format WAR (*Web Archive*). Representa una aplicació web empaquetada.

La taula anterior no és una llista exhaustiva. Hi ha molts elements de Java EE que no hi trobem i que necessitarem per a modelitzar un sistema complet.

L'objectiu d'aquest subapartat no és definir un perfil complet per a Java EE, sinó que copseu la importància que té disposar d'un perfil per a la tecnologia concreta i com ens ajudarà per a especificar la fase de disseny que permetrà passar de l'especificació independent de tecnologia a la implementació del sistema amb Java EE.

3.2. Fase de disseny

Per a fer el disseny partirem del diagrama de components i el seu diagrama de desplegament. Després els adaptarem, tenint en compte les particularitats de la tecnologia escollida, fins aconseguir uns diagrames de components que modelitzin el més concretament possible la implementació d'aquests components amb Java EE i en reflecteixin les particularitats i les decisions de disseny.

Vegeu també

Heu vist els diagrames de components i de desplegament en el mòdul "Desenvolupament de programari basat en components" d'aquesta assignatura.

Per exemple, un component que modelitzi un **gestor de transferències** es pot implementar amb Java EE fent servir un EJB de sessió sense estat que faci de façana i una entitat JPA per a fer les dades persistents. El pas de l'especificació a la implementació amb Java EE refinarà el component *GestorTransferencies* del diagrama arquitectònic de components fins a obtenir un diagrama de components de gra més prim amb tots els detalls d'implementació d'aquest component.

Després de la fase de disseny, passar del diagrama de components d'implementació a codi ha de ser una tasca força senzilla (pràcticament una traducció). En aquest sentit s'està avançant força en àmbits com l'MDA[34] per a aconseguir eines que facin una traducció automàtica del model aconseguit a codi.

(34)De l'anglès *Model Driven Architecture*.

Vegeu també

Vegeu el mòdul introductori de l'assignatura "Enginyeria del programari" per a més informació sobre MDA.

També veurem com es pot representar la distribució física dels components amb Java EE i refinarem els diagrames de desplegament per a incloure-hi els conceptes d'empaquetament i desplegament que hi ha a la plataforma Java EE.

Per exemple, tots els artefactes que formen la implementació del component *GestorTransferencies* s'empaqueten en un fitxer anomenat *GestorTransferencies.jar* que es desplega en els dos contenidors d'EJB que formen part del clúster de l'aplicació de banca electrònica de l'entitat financera que ens ha encarregat el desenvolupament.

3.2.1. Diagrames considerats

Tot i que en la fase de disseny de l'aplicació cal modelitzar concretament tant els aspectes dinàmics com els aspectes estàtics del sistema, en aquest subapartat ens centrarem a veure el pas dels diagrames de components que descriuen els components arquitectònics del sistema a diagrames de **components** o de **classes** que descriuran el sistema en una tecnologia i també veurem els diagrames de desplegament.

3.2.2. Decisions de disseny

En la fase de disseny es fan refinaments successius sobre els diagrames de components arquitectònics fins a obtenir uns diagrames de components d'implementació. En aquests refinaments caldrà prendre algunes decisions de disseny que marcaran la implementació final.

L'objectiu d'aquest subapartat no és fer un recull exhaustiu de totes les decisions que cal considerar a l'hora de dissenyar una aplicació Java EE; només en mostrarem algunes de les més habituals.

Com ja hem vist, Java EE defineix un model d'aplicació multicapa amb *n* nivells. Una possible via per a atacar les decisions de disseny és fer-ho amb una divisió dels components de l'aplicació segons la capa en la qual es troben. Si procedim així, algunes decisions que cal tenir en compte en moltes aplicacions típiques Java EE són:

Decisions

Noteu que moltes de les decisions que cal avaluar les hem comentades en els apartats en què descrivíem les decisions de disseny per a les capes de presentació, negoci i integració.

1) Decisions de disseny de la capa de presentació:

- Quin model farem servir per a implementar la capa de presentació?
- Quins components implementaran la lògica de presentació i quins l'aspecte gràfic?
- Farem servir algun *framework* per a aquesta capa?
- Com accedirem a la capa de negoci?

2) Decisions de disseny de la capa de negoci:

- A quins components/funcionalitats s'accedirà localment i quins estaran distribuïts?
- Seguirem algun patró de disseny com, per exemple, una façana?
- Els components de negoci tenen estat?
- Els components de negoci tenen comportament síncron o asíncron?

3) Decisions de disseny de la capa de persistència:

- Com implementarem la persistència?
- Farem servir JDBC directament? Seguirem algun patró de disseny?
- Farem servir algun tipus d'eina de mapatge objecte-relació de tercers?
- Farem servir entitats JPA?

3.2.3. Exemple

Per a veure com s'apliquen algunes de les decisions de disseny que hem esmentat en el subapartat anterior i com fan evolucionar els diagrames de components fins arribar gairebé a la implementació, suposem que volem fer el disseny d'un component que permet als clients d'un banc en línia fer operacions amb els seus comptes.

En un mòdul anterior, aquest component es va representar com es presenta en la figura 29 (en el nivell més alt d'abstracció).

Vegeu també

El component per a fer operacions amb comptes correspon al component *OperacionsBàsiques* que heu vist en l'exemple del component ATM, en l'apartat "Composició tardana, vinculació dinàmica i delegació" del mòdul "Desenvolupament de programari basat en components".

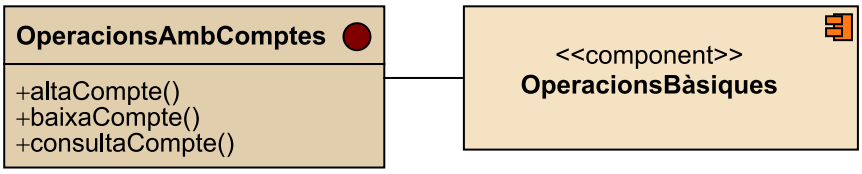

Figura 29. Component independent de la tecnologia

La primera decisió que prenem és que el component s'implementarà amb Java EE i, per tant, tindrà una part de presentació (per a permetre la interacció dels usuaris amb el servei del component), una de negoci (la funcionalitat del component) i una d'integració (per a poder relacionar-se amb el suport persistent on es desen de manera permanent les dades).

Nota

Per a mostrar algunes de les decisions de disseny que cal prendre suposarem que el component *OperacionsBàsiques* mateix conté la capa de presentació i la capa d'integració.

Això no era així en l'exemple del component ATM, en què la part de presentació i d'integració les implementaven altres components.

Tingueu en compte que estem suposant que el component ***OperacionsBàsiques*** té part de presentació i part d'integració.

En l'arquitectura general els components encarregats de cada "capa" són diferents i caldria aplicar el procés de disseny corresponent a cada un.

Podem refinar, doncs, el diagrama anterior per a obtenir el de la figura 30.

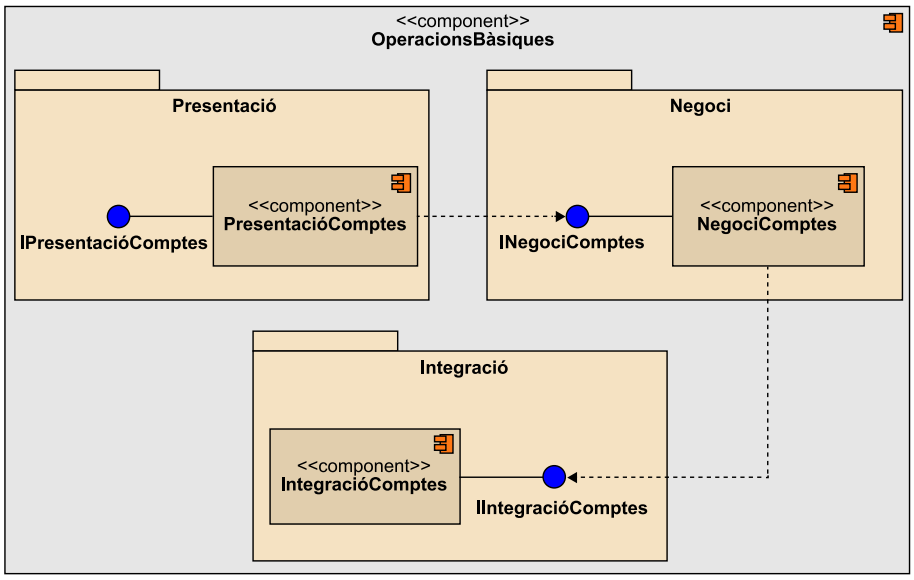

Figura 30. Divisió del component amb presentació, negoci i integració

En aquest punt hem dividit el component original en tres subsistemes de responsabilitats més delimitades:

1) Component de presentació. Permet als usuaris interactuar amb la lògica de negoci, proporciona la interfície d'usuari del component.

2) Component de negoci. Conté la lògica de negoci de les operacions que els usuaris poden fer amb els comptes, com altes, baixes, consultes, etc.

3) Component d'integració. Conté la lògica que permet al component de negoci interactuar amb els magatzems de dades.

A partir d'aquí agafarem el component de cada capa i prendrem les decisions de disseny que considerem oportunes (alguns cops aquestes decisions de disseny estaran marcades pels requisits del sistema). Vegem-ne un exemple.

Component de presentació

Suposem que decidim utilitzar una estructura Model-2 seguint un patró de disseny MVC[35]. Si seguim aquest patró de disseny i suposem que el model estarà implementat en el component de negoci, podem dividir el component de presentació en el component que farà de controlador i el component que farà de vista.

[35]Sigla de *Model Vista Controlador*.

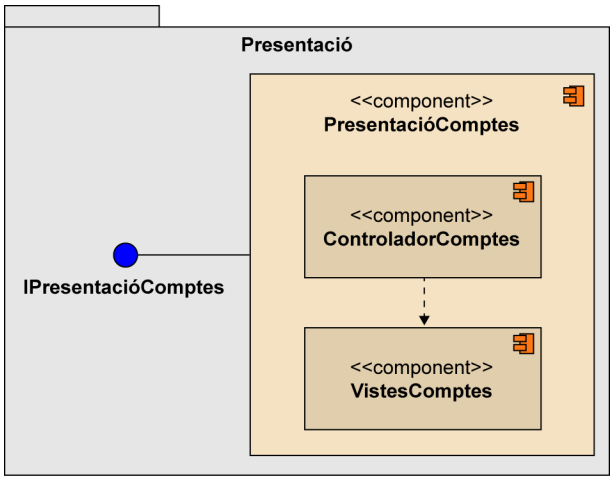

Figura 31. Component de presentació

Si suposem que el component permet fer les operacions ***altaCompte***, ***baixaCompte*** i ***consultaCompte*** i prenem les decisions de disseny següents:

- Utilitzarem el *framework Java Server Faces*.
- El controlador serà el *servlet FacesServlet*.
- Les accions estaran definides amb *Managed Bean*.
- Implementarem les vistes amb *facelets*.
- Cada operació tindrà una vista de sortida.

Podem refinar el diagrama de la figura 31 fins arribar al que presenta la figura 32.

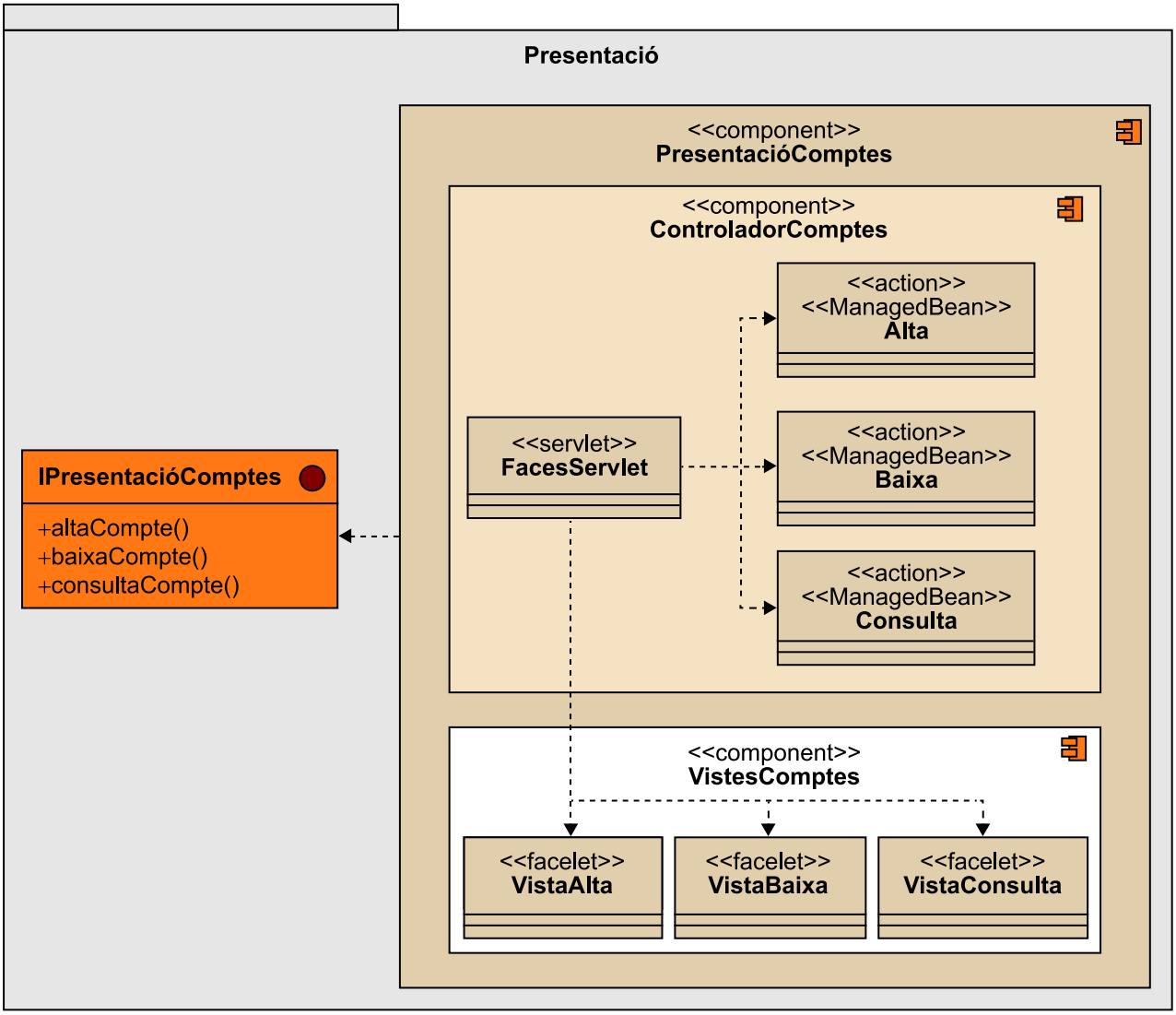

Figura 32. Refinament del component de presentació

Tingueu en compte que ens hem inventat els estereotips *<<servlet>>*, *<<facelet>>*, *<<ManagedBean>>* i *<<action>>* que no teníem en el perfil UML definit en el document de *Superestructura* que hem agafat com a referència.

Component de negoci

Suposem que volem donar accés remot al component que modelitza els comptes. Amb aquesta premissa decidim que la millor alternativa per a la capa de negoci és un component EJB. Suposem també que les operacions que farem amb els comptes no tenen estat, que les transaccions les gestionarà el contenidor i que les peticions dels client són sempre síncrones. De les decisions de disseny anteriors deduïm que el component de negoci es pot implementar amb un EJB de sessió sense estat amb accés remot que segueix el patró façana i el podem representar amb el diagrama de classes que presenta la figura 33.

Codi

Noteu que la traducció d'aquest diagrama a codi és directa i és podria automatitzar.

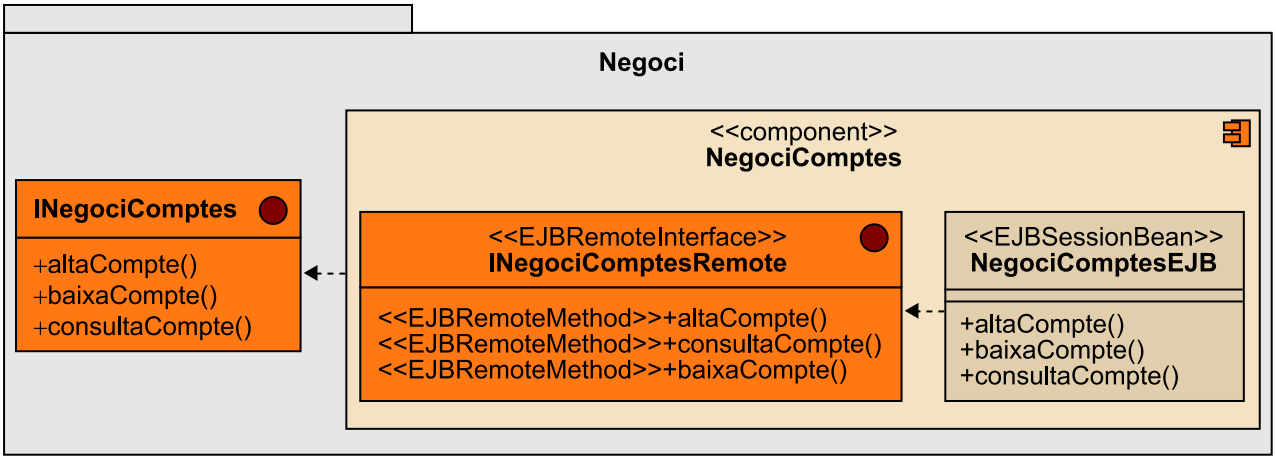

Figura 33. Component de negoci

Component d'integració

Suposem que hem d'operar amb comptes que es troben en una taula d'una base de dades relacional i que hi hem de fer les operacions de consulta, alta i baixa. Amb aquestes premisses, podríem decidir implementar el component d'integració amb entitats JPA (encara que tenim altres opcions). Per a aquest cas simple, només tenim una relació $1 \rightarrow N$ entre cada compte i els seus titulars. Suposem també que a la capa de persistència només s'accedirà des de la capa de negoci i aquest accés es farà de manera local. Si tenim en compte totes aquestes decisions podríem obtenir el diagrama de components que presenta la figura 34.

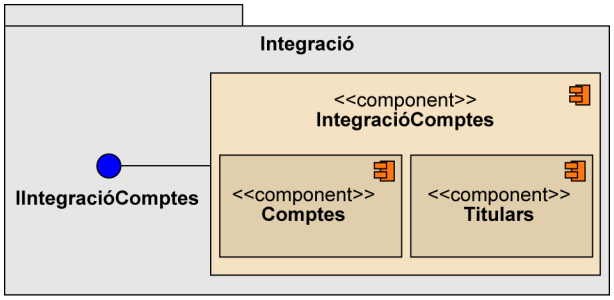

Figura 34. Component d'integració

I aplicant el perfil Java EE queda com es presenta en la figura 35.

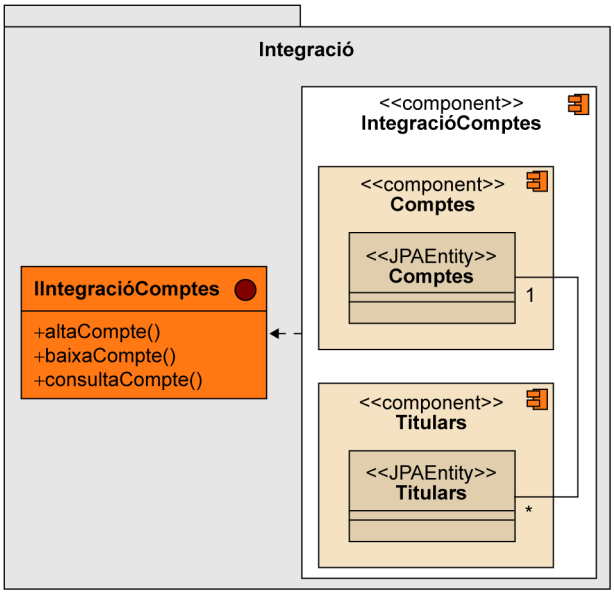

Figura 35. Refinament del component d'integració

Suposem que decidim empaquetar la capa de presentació amb una aplicació web anomenada, per exemple, ***comptes.war***, i empaquetar els components que formen la capa de negoci i d'integració en un únic fitxer anomenat ***comptes.jar***.

Suposem també que despleguem la part web en una màquina que fa de contenidor web i la part de negoci i persistència en una altra màquina amb un contenidor d'EJB. El diagrama de desplegament podria quedar com es mostra en la figura 36.

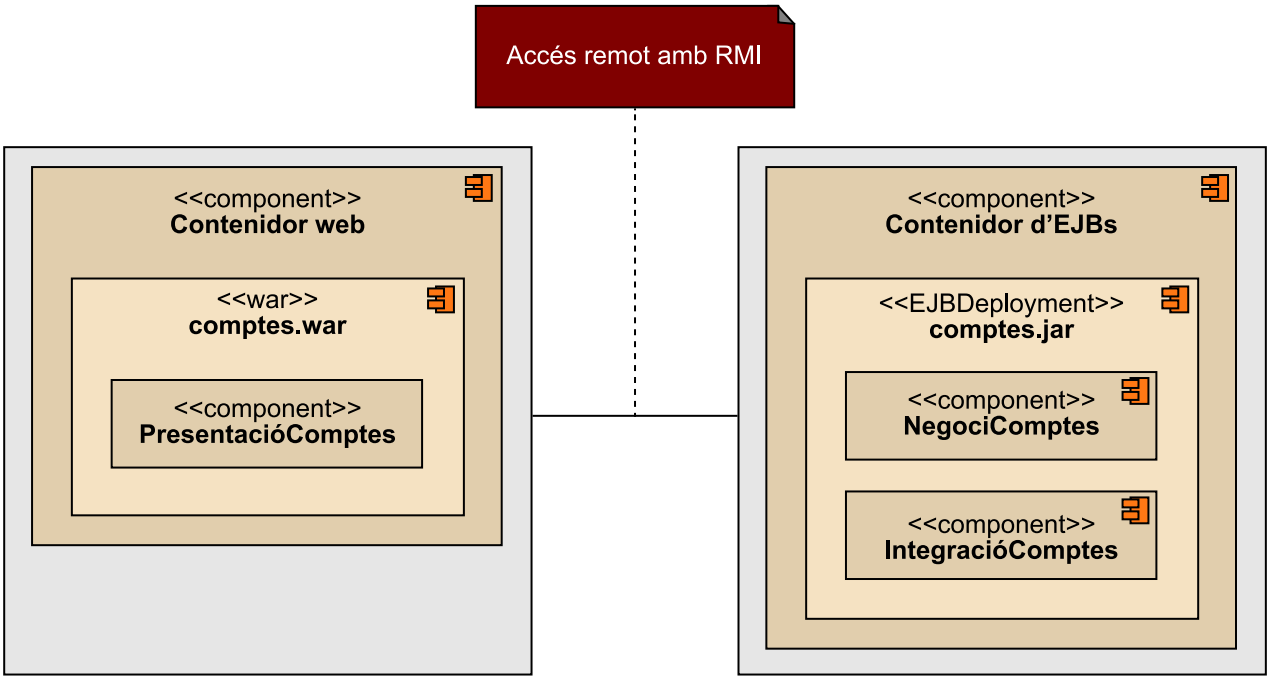

Figura 36. Diagrama de desplegament
Noteu que aquest diagrama es pot refinar encara molt, fins a incorporar tots els detalls rellevants del desplegament de l'aplicació.

Un cop arribats a aquest punt, i després d'aplicar totes les decisions de disseny necessàries, el disseny de l'aplicació ja està complet i conté tota la informació necessària per a procedir a implementar-la.

Resum

Aquest mòdul ha proporcionat una visió a alt nivell de Java EE, una opció tecnològica per a implementar sistemes distribuïts basats en components. Java EE és una de les plataformes empresarials més utilitzades actualment.

Al llarg del mòdul hem definit **Java EE** i molts dels elements que en formen part, hem vist el model d'aplicacions que defineix Java EE, els components que defineix l'especificació i com es poden combinar per a formar una aplicació.

Hem dedicat especial atenció a la **divisió per capes** que promou el model d'aplicacions Java EE, que en defineix 5: client, presentació, negoci, integració i EIS. Ens hem fixat també en la capa de servidor, que engloba les capes de presentació, negoci i integració.

Hem definit els conceptes de contenidor, component, servei i servidor d'aplicacions restringits a Java EE i hem presentat els tipus de components que defineix l'especificació: en la capa de presentació hem vist els *servlets* i el *framework Java Server Faces*, en la capa de negoci dos tipus d'EJB (EJB de sessió i de missatge) i en la capa d'integració les entitats JPA. Els hem situat dins l'arquitectura Java EE i n'hem analitzat els pros i els contres.

També hem vist les possibles arquitectures en les quals es pot emmarcar una aplicació Java EE i quins components cal utilitzar en cada cas.

El darrer apartat del mòdul ens ha servit per a presentar les bases del **disseny d'aplicacions Java EE**, amb tot un seguit de recomanacions i aspectes per tenir en compte.

Java EE és una plataforma molt complexa i el que hem vist és només una visió general del seu potencial. Aquest mòdul us ha de servir com a introducció i per a entendre els principis bàsics que hi ha darrere de Java EE.

Activitats

1. Feu una taula que relacioni les capes d'una aplicació Java EE, la seva funcionalitat i els components que hi podem trobar.

2. Expliqueu breument en què consisteix la tecnologia AJAX i com creieu que pot afectar la capa de presentació de les aplicacions Java EE.

3. Codifiqueu un exemple d'un EJB de sessió sense estat que digui "Hello world!" amb EJB 2.x i amb EJB 3.x i feu-ne una comparativa avaluant, entre altres aspectes, la complexitat en el desenvolupament, el nombre d'artefactes que cal crear en cada cas, les restriccions que cal imposar a l'especificació, etc.

4. Codifiqueu un client Java que accedeixi a un EJB de sessió sense estat que ofereix un mètode *sayHello()* utilitzant EJB 2.x i EJB 3.x. Comenteu per a cada un dels dos casos la seqüència de crides que es produeixen.

5. Expliqueu què són les *anotacions* i com s'utilitzen en l'especificació d'EJB 3.x. Raoneu si us semblen útils.

6. Expliqueu què és la injecció de dependències i com s'utilitza en l'especificació de Java EE. Raoneu si us semblen útils aquests conceptes.

7. Feu el disseny del component que permetia operar amb els comptes tenint en compte que ara decidim que els components de negoci estiguin a la mateixa màquina que els components de presentació i no utilitzarem *Java Server Faces* per a la capa de presentació ni entitats JPA per a la capa de persistència.

8. Feu el disseny del component que permet als clients d'un banc en línia fer operacions amb els seus comptes que hem analitzat en el subapartat "Exemple" sense utilitzar JSF per a la capa de presentació i amb EJB 2.x tant per a la capa de negoci com per a la capa d'integració.

Nota

Tot i que no és habitual proporcionar les solucions a les activitats proposades, a causa de la seva importància, us proporcionem un esquema amb la resolució de l'activitat número 8. La resta d'activitats es deixen sense solucionar com a exercici d'estudi.

Exercicis d'autoavaluació

1. Raoneu per què no és del tot correcte dir la frase "Hem de comprar el Java EE".

2. Relacioneu en un paràgraf els conceptes *Java EE*, *component*, *contenidor*, *servei* i *servidor d'aplicacions*.

3. Expliqueu breument sota quin estil arquitectònic es defineix el model d'aplicacions Java EE.

4. Expliqueu els diferents tipus de *managed beans* que es poden definir en una aplicació web amb *Java Server Faces* i com es relacionen amb el patró MVC.

5. Què creieu que passaria si no hi hagués una fase de disseny entre l'especificació i la implementació en la construcció d'una aplicació?

Solucionari

Activitats

8. Les restriccions de disseny que ens imposen ara no canvien la divisió del component en la capa de presentació, negoci i integració (figura 30).

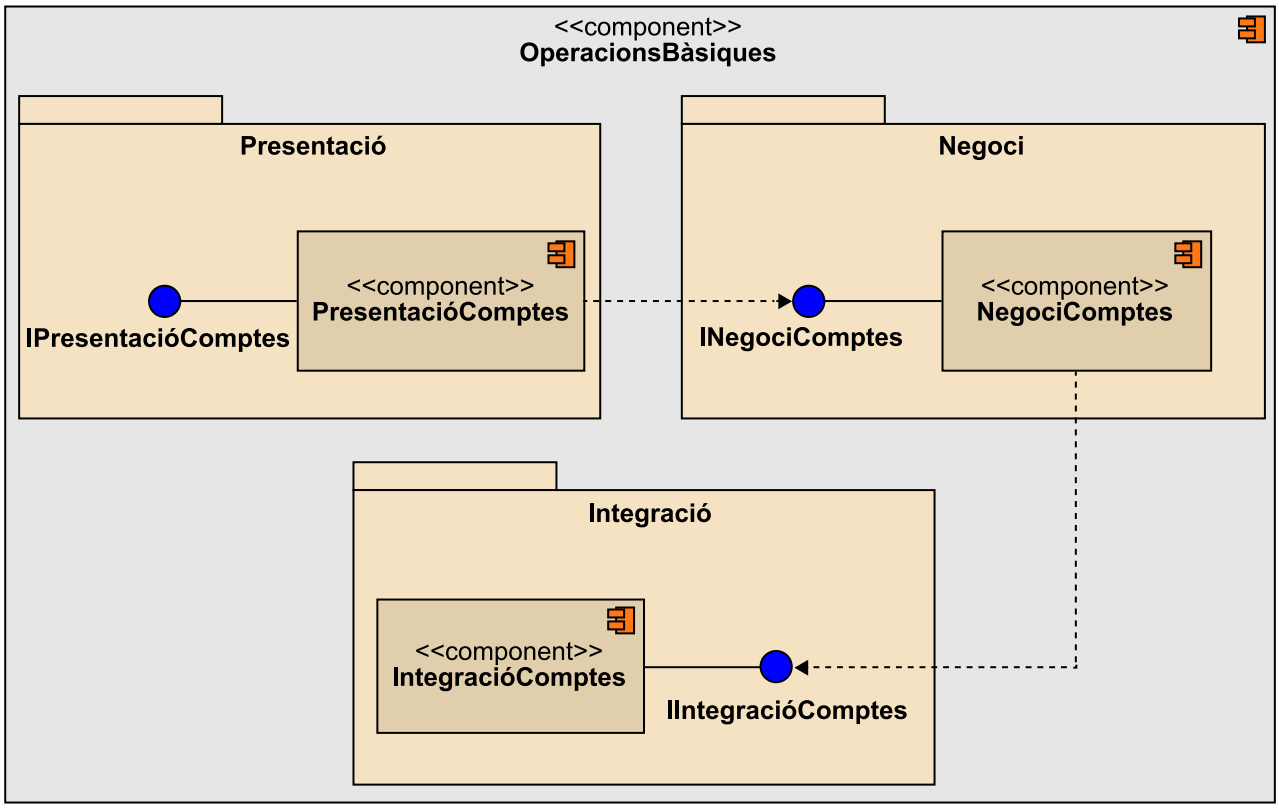

Figura 30. Divisió del component en presentació, negoci i integració

Component de presentació. No podem utilitzar JSF, i per tant les vistes s'implementaran amb JSP i no tindrem *managed beans* per a implementar les accions. També haurem d'implementar nosaltres la el *servlet* que faci de controlador. Amb tot l'anterior, el component de la capa de presentació ens queda com mostra la figura 37, un cop aplicat el perfil Java EE.

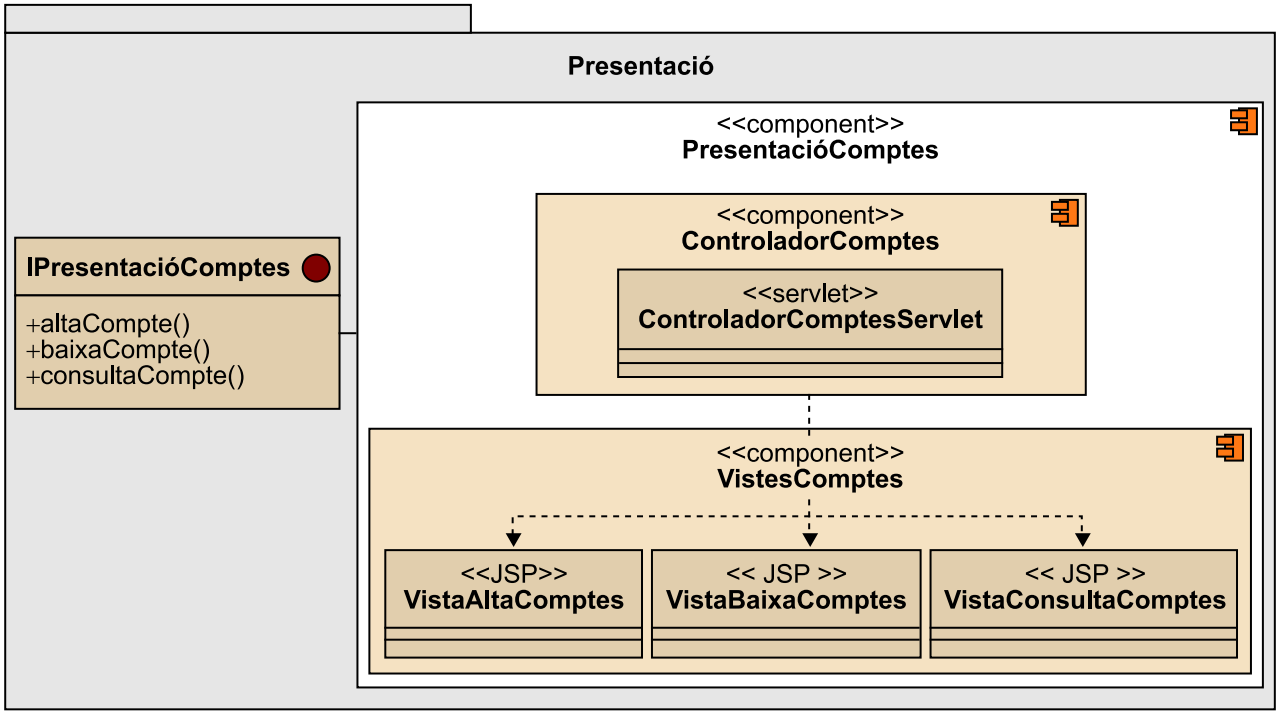

Figura 37. Component de presentació sense JSF

Component de negoci. El component de negoci ha d'utilitzar EJB 2.x, i el podem representar tal com es presenta en la figura 38.

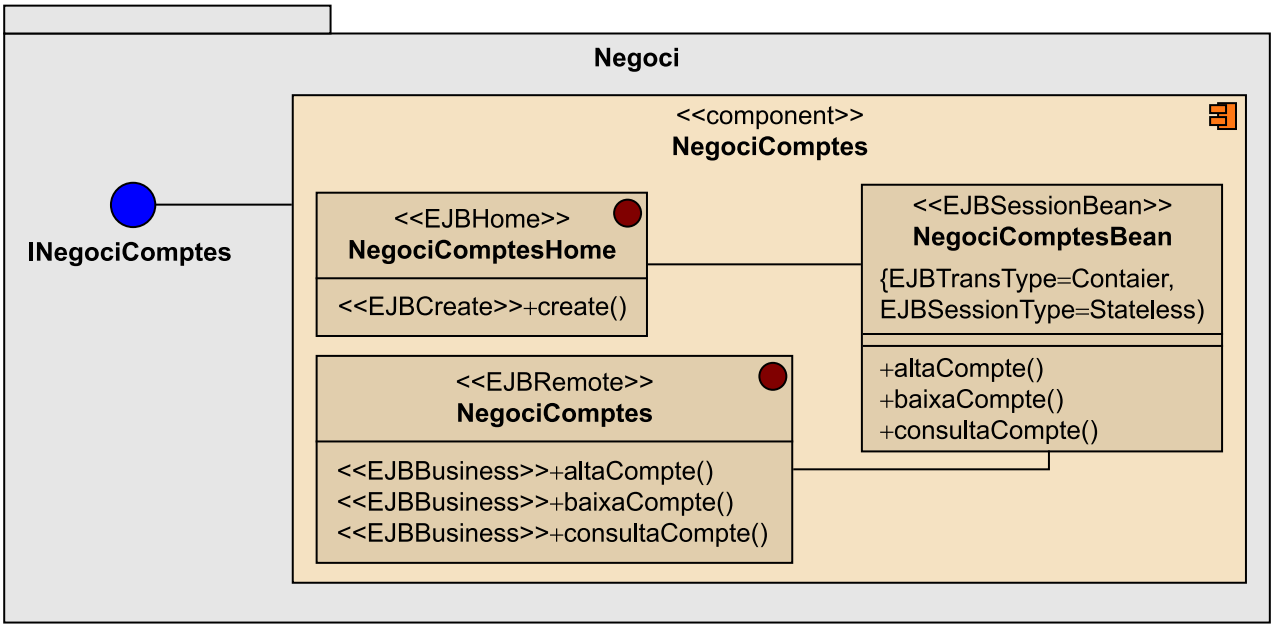

Figura 38. Component de negoci amb EJB 2.x

Component d'integració. En aquest component hem de canviar les entitats JPA per EJB d'entitat (encara que tenim altres opcions). Optem també per utilitzar CMP per a gestionar la persistència del component i, per a aquest cas simple, només tenim una relació $1 \rightarrow N$ entre cada compte i els seus titulars. Suposem també que a la capa de persistència només s'accedirà des de la capa de negoci i aquest accés es farà de manera local. Si tenim en compte totes aquestes decisions podríem obtenir el diagrama de components que presenta la figura 39.

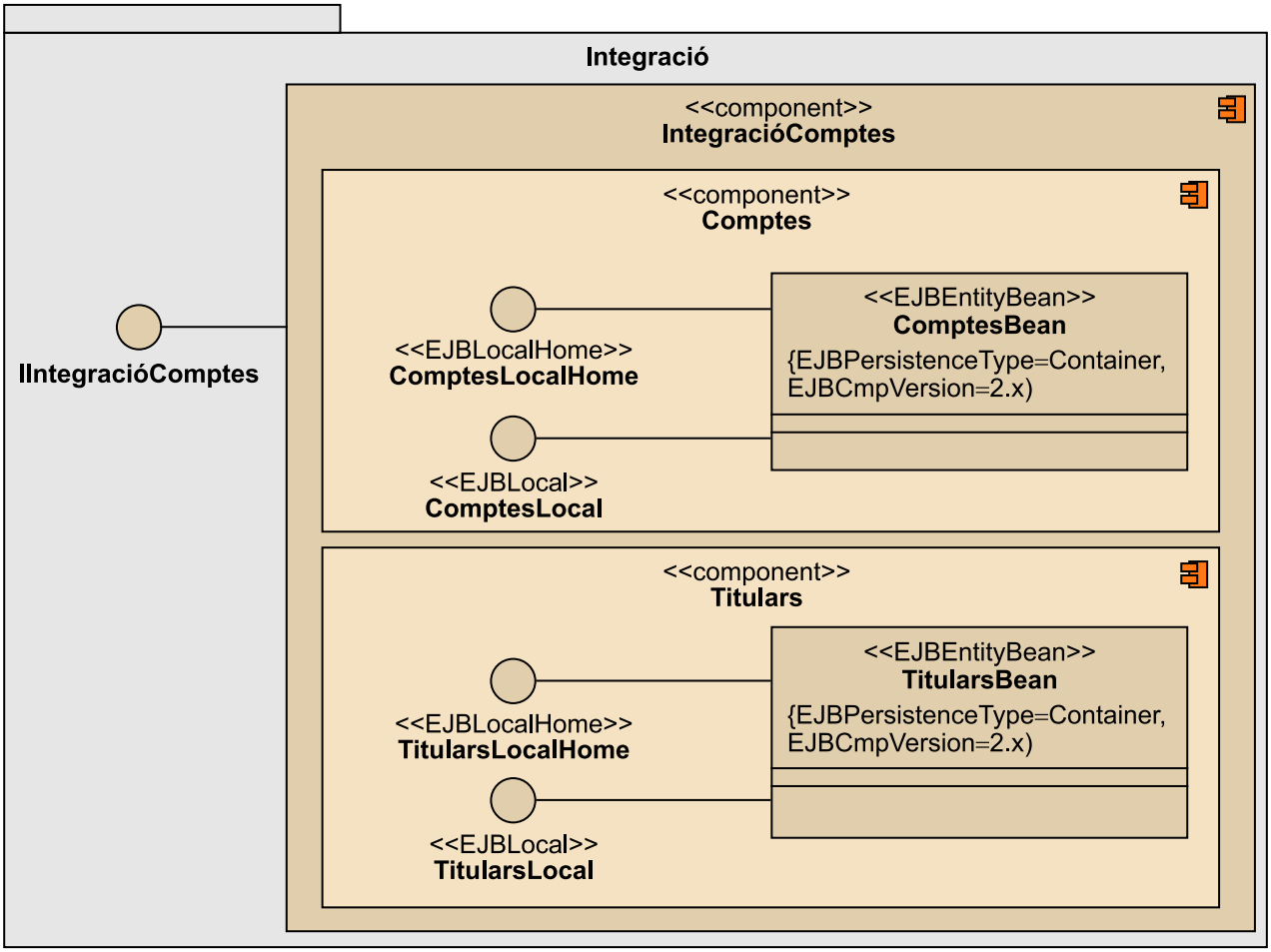

Figura 39. Component d'integració amb EJB d'entitat

Exercicis d'autoavaluació

1. La frase "Hem de comprar el Java EE" no és del tot correcta perquè Java EE **no** és un producte. Aquesta és una confusió molt freqüent. Java EE no és un producte concret que distribueix Oracle, sinó una plataforma de desenvolupament empresarial formada per un model de programació, un conjunt d'especificacions que han d'implementar els servidors d'aplicacions (plataforma Java EE), uns tests de compatibilitat i una implementació de referència. De tot l'anterior, l'únic que és un producte de programari és la implementació de referència. El que es compra o s'instal·la per a poder desplegar aplicacions Java EE són implementacions de la plataforma (servidors d'aplicacions, com JBoss i IBM WebSphere).

2. Hi ha infinitat de frases amb les quals es poden relacionar els conceptes *Java EE*, *component*, *contenidor*, *servei* i *servidor d'aplicacions*, com per exemple: una aplicació Java EE estarà formada per un conjunt de components que s'assemblen i es despleguen en contenidors per a oferir als usuaris la funcionalitat desitjada; els contenidors ofereixen als components accés als serveis que defineix Java EE, i els servidors d'aplicacions implementen aquests serveis.

3. El model d'aplicacions que defineix Java EE s'emmarca dins un estil arquitectònic heterogeni que combina característiques de diferents estils, i se centra en un estil client-servidor, basat en components i organitzat en capes o nivells.

4. Podem distingir entre cinc tipus diferents de *managed beans*:

- *Model managed bean*: implementa el concepte de model del patró MVC. Típicament és una classe POJO que segueix les convencions dels *Java beans* amb *getters* i *setters* per als atributs. Modelitza les entitats del domini.
- *Backing managed bean*: participa en el concepte de vista del patró MVC. Igual que els *model managed beans*, segueix les convencions dels *Java beans*, però les seves propietats es vinculen a propietats de la vista i no a propietats del model. Normalment defineixen una relació 1:1 amb les vistes.
- *Controller managed bean*: participa en el concepte de controlador del patró MVC. Aquests *beans* són cridats pel *servlet* que fa de controlador, implementen les accions de negoci i tornen el control a JSF per a executar la lògica de navegació. Típicament tenen mètodes

d'acció i implementen el patró de disseny *Command* dins l'arquitectura JSF. En temps de disseny decorarem aquests *beans* amb l'estereotip *<<action>>*.

- *Support managed bean*: participen en el concepte de vista del patró MVC, i donen suport a una o diverses vistes.
- *Utility managed beans*: podeu veure aquest tipus de *beans* com a classes d'utilitat que també donen suport a les vistes.

5. La fase de disseny permet que el salt entre l'especificació independent de la tecnologia i la implementació amb una tecnologia concreta sigui el més petit possible. En aquesta fase es refinen els diagrames de components independents de la tecnologia per a introduir-hi les particularitats de la tecnologia concreta escollida per a fer la implementació. Així, els desenvolupadors, pràcticament, només hauran de traduir els components que surtin del disseny (ara ja amb una tecnologia concreta) a codi.

Glossari

assemblament *m* Procés en què s'empaqueta en un format estàndard tot el que forma un component o una aplicació per a poder fer el desplegament.

contenidor *m* Peça de programari en què es despleguen els components i que ofereix accés als serveis que proporciona la plataforma.

desplegament *m* Procés d'instal·lació del component o l'aplicació al contenidor corresponent.

EJB de missatge *m* Component sense estat, de servidor, transaccional, que serveix per a respondre a missatges asíncrons.

EJB de sessió *m* Component que serveix per a implementar la lògica de negoci de les aplicacions Java EE.

Enterprise Java Bean *m* Component de negoci distribuït, desenvolupat en Java, que compleix unes especificacions i, per tant, es pot desplegar en qualsevol contenidor d'EJB d'un servidor d'aplicacions compatible amb la plataforma Java EE.
Sigla **EJB**

entitat JPA *f* Component que representa les dades persistents del model de negoci de l'aplicació. Els podeu veure com una representació Java, en memòria i en forma d'objecte, de la informació emmagatzemada en algun mitjà persistent (com una base de dades relacional).

Java EE *m* Plataforma de desenvolupament empresarial que defineix un estàndard per al desenvolupament d'aplicacions empresarials multicapa.

Java Server Faces *m* *framework* MVC de desenvolupament de la capa web per a aplicacions Java EE.

Java Server Page *m* Document de text que s'executa com a *servlet*, però que permet una aproximació més natural a la creació de contingut estàtic.
Sigla **JSP**

Perfil UML *m* Mecanisme que proporciona UML per a estendre'n la sintaxi i la semàntica i poder modelitzar les particularitats d'un domini o tecnologia.

Plain Old Java Interface *m* Interfície Java normal.
Sigla **POJI**

Plain Old Java Object *m* Classe Java normal.
Sigla **POJO**

servlet *m* Classe Java que rep peticions HTTP i genera contingut dinàmic en resposta a aquestes peticions.

servidor d'aplicacions *m* Peça de programari que implementa els serveis que ofereixen els contenidors als components.

Bibliografia

Bibliografia bàsica

Alur, D.; Crupi, J.; Malks, D. (2003). *Core Java EE Patterns: Best Practices and Design Strategies* (2a. ed). Upper Saddle River: Prentice Hall.

Johnson, R. (2004). *Expert One-on-One Java EE Development without EJB*. Indianapolis: Wiley Publishing.

Roman, E.; Ambler, S.; Jewell, T. (2006). *Mastering Enterprise JavaBeans 3.0* (2a. ed.). Nova York: Wiley Computer Publishing.

Singh, I.; Stearns, B.; Johnson, M. (2002). *Designing Enterprise Applications with the Java EE Platform* (2a. ed.). Boston: Addison-Wesley.

Bibliografia complementària

Ahmed, K. Z.; Cary, E. U. (2001). *Developing Enterprise Java applications with Java EE™ and UML*. Indianapolis: Addison-Wesley.

Fowler, M. (2003). *Patterns of Application Architecture*. Addison-Wesley.

Johnson, R. (2003). *Expert One-on-One Java EE Design and Development*. Indianapolis: Wiley Publishing.

Tate, A.; Clark, M.; Lee, B.; Linskey, P. (2003). *Bitter EJB*. Greenwich: Manning Publications.

Torre, C. de la; Zorrilla, U.; Calvarro, J.; Ramos, M. A. (2010). *Guía de Arquitectura N-Capas Orientada al Dominio con .NET 4.0*. Madrid: Krassis Consulting.

Especificacions oficials

Java Community Process (2009). *Java™ Platform Enterprise Edition (Java EE) 6*. Disponible a http://jcp.org/aboutJava/communityprocess/pr/jsr316/index.html.

Object Management Group (2011). *Unfied Modeling Language™ (UML®)*. Disponible a http://www.omg.org/spec/UML.

Oracle. *Java EE 6 Technologies*. Disponible a http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html.

Enllaços

Java Hispano. Portal de notícies de Java en castellà: http://www.javahispano.org

Java Lobby. Portal de notícies relacionades amb Java EE: http://www.javalobby.org/

OMG (Object Management Group). Pàgina principal de l'OMG: http://www.omg.org

Oracle. Pàgina oficial de Java EE: http://www.oracle.com/technetwork/java/javaee/overview/index.html

The Server Side. Portal de notícies relacionades amb Java EE: www.theserverside.com