Alkalmazott informatika sorozat

Szoftverfejlesztés Java EE platformon

Szerkesztette: Imre Gábor

SZAK KIADÓ

Elektronikus kiadás ISBN 978-963-9863-55-2

Balogh Péter – Berényi Zsolt – Dévai István
Imre Gábor – Soós István – Tóthfalussy Balázs

Szoftverfejlesztés Java EE platformon

Balogh Péter – Berényi Zsolt
Dévai István – Imre Gábor
Soós István – Tóthfalussy Balázs

Szoftverfejlesztés Java EE platformon

Változatlan utánnyomás
elektronikus kiadás

2016

Szoftverfejlesztés Java EE platformon

Balogh Péter – Berényi Zsolt – Dévai István – Imre Gábor – Soós István – Tóthfalussy Balázs

Változatlan utánnyomás, elektronikus kiadás
Alkalmazott informatika sorozat

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék
Alkalmazott Informatika Csoport

Sorozatszerkesztő: Dr. Charaf Hassan
Lektor: Dr. Gál Tibor
Borítóterv: Péteri Szilárd és Kovács Tibor
Grafikai munka kivitelezője: Flórián Gábor (Typoézis Kft.)

ISBN 978-963-9863-55-2 (pdf)
ISSN 1785-363X

A szöveg helyességét és az elválasztásokat a MorphoLogic Helyesek nevű programjával ellenőriztük.

SZAK Kiadó Kft. ■ Az 1795-ben alapított Magyar Könyvkiadók és Könyvterjesztők Egyesülésének a tagja ■ 2060 Bicske, Diófa u. 3. ■ www.szak.hu ■ e-mail: info@szak.hu ■ Kiadóvezető: Kis Ádám, e-mail: adam.kis@szak.hu Főszerkesztő: Kis Balázs e-mail:balazs.kis@szak.hu

Tartalomjegyzék

Első fejezet
Bevezetés

A Java nyelv története 1991-ig nyúlik vissza, a nyilvánosság számára azonban csak 1995-től vált ismertté. Az azóta eltelt évek során a nyelv, illetve a hozzá kapcsolódó technológiák dinamikus fejlődésen mentek át, miközben egyre többféle feladat megoldására váltak alkalmassá, így használatuk egyre szélesebb körben elterjedt. Mindez indokolttá tette, hogy a Java platformnak 3 ún. „kiadása" (edition) jelenjen meg, melyek eltérő típusú szoftverek fejlesztését teszik lehetővé. Ezen kiadások közül a **Standard Edition** (Java SE, korábban J2SE) célozza meg a hagyományos desktop alkalmazások, illetve appletek készítését, amelyekre a Java már a kezdetektől fogva alkalmas volt. A korlátozott erőforrásokkal rendelkező, jellemzően mobil eszközökre történő fejlesztést a **Micro Edition** (Java ME, korábban J2ME) támogatja. Ennek a kiadásnak jellemzője, hogy a Standard Editionben elérhető osztályoknak és interfészeknek csak egy viszonylag kisebb részhalmazát tartalmazza, viszont tartalmaz mobilspecifikus API-kat (alkalmazásprogramozói interfészeket), amelyek csak Micro Edition környezetben érhetők el. Az eltérő képességű eszközök kezelésére különböző profilokat implementáltak. Végül a legnagyobb méretű kiadás, egyben könyvünk témája, a **Java Enterprise Edition** (Java EE, korábban J2EE) elosztott, sok felhasználóval rendelkező, vállalati méretű szoftverrendszerek kialakításakor vethető be. A Java EE-nek részhalmaza a Java SE, vagyis az Enterprise technológiák mellett minden standard Java API is rendelkezésre áll.

A kiadások rövidítésében látható, hogy azokban korábban szerepelt a 2-es szám. Ennek oka, hogy a Javát az 1.2-es verzióban bekövetkezett jelentős változások (1998) óta Java 2-nek hívták. Így innentől kezdve az 1.2-es, 1.3-as (2000) és 1.4-es (2002) standard, illetve a rájuk épülő enterprise kiadások rövidített neve J2SE/J2EE 1.2/1.3/1.4 volt. A 2004-es Java 1.5 azonban ismét hozott annyi újdonságot a nyelvbe, hogy kiérdemelte a Java 5 nevet, mivel pedig így a 2-es szám fölöslegessé vált, Java SE/EE 5 lett a hivatalos rövidítés.

A Java EE egymondatos meghatározása az alábbi lehetne: architektúra **vállalati méretű** alkalmazások fejlesztésére, a Java nyelv és internetes technológiák felhasználásával. Joggal tehetjük fel a kérdést: a standard Java által nyújtott lehetőségek miért nem elegendőek? Nagyméretű információs rendszerek fejlesztése során számos, jellegében hasonló követelmény merül fel (pl. skálázhatóság, biztonság, más rendszerekkel való integrálhatóság). A Java EE oly módon támogatja a vállalati méretű rendszerek fejlesztését, hogy ezen gyakori, együtt fellépő problémákra nyújt globális megoldást. Mégpedig azáltal, hogy olyan futási környezetet, egyfajta keretrendszert biztosít az általunk fejlesztett alkalmazások számára, amelyben különféle szolgáltatások használhatók fel az említett igények kielégítésére.

Egy konkrét példán keresztül vizsgáljuk meg közelebbről ezeket a követelményeket! Képzeljük el, hogy egy banki alkalmazást kell fejlesztenünk, amely teljes körű banki ügyvitelt valósít meg (folyószámla-vezetés, betétek kezelése, hitelezés stb). Az ügyfelek webes felületen keresztül is igénybe vehetik a szolgáltatásokat, a banki alkalmazottak számára viszont hagyományos desktop alkalmazás áll rendelkezésükre. Milyen problémákkal kell megbirkózni egy ilyen feladat megoldása során?

- **Perzisztencia** (állandóság, folyamatosság: a latin „persistere" szóból; számítógépes vonatkozásban adatfennmaradás jelentéssel). A rendszer működéséhez szükséges adatokat (számlák, betétek, ügyfelek stb.) perzisztens módon kell tárolni, vagyis biztosítani kell, hogy az adatok a program leállítása után is megmaradjanak. Erre a feladatra leggyakrabban relációs adatbázisokat alkalmaznak. Az alkalmazásunkban tehát meg kell oldanunk relációs adatbázisok elérését. Nyilvánvaló, hogy olyan adatelérési technológiát várunk el, amely elfedi előlünk az egyes adatbázis-kezelő rendszerek alacsony szintű, gyártóspecifikus részleteit (pl. alkalmazott hálózati protokoll), továbbá összeegyezteti az objektumorientált szemléletmód és a relációs adatbázisok alapelveit.
- **Többszálúság.** A rendszerünket egyidejűleg több felhasználó is elérheti. Több kérés párhuzamos kiszolgálására kézenfekvő megoldás, hogy több szál dolgozza fel a kéréseket. Sokat segít a fejlesztőnek, ha olyan futási környezetre fejleszthet, ami a szálkezelés részleteinek legalább egy részét elrejti előle.
- **Tranzakciókezelés.** Ha több kliens konkurensen akar hozzáférni ugyanazon adatokhoz, ügyelni kell arra, hogy az adatok mindig konzisztens állapotban legyenek. Lehetőséget kell nyújtani arra, hogy bizonyos összetartozó módosításoknak vagy mindegyike sikerüljön, vagy egyike sem. Az egyes kliensek szempontjából pedig kívánatos, hogy úgy lássák az adatokat, mintha egyedül használnák a rendszert. Ezekre a problémákra a tranzakciókezelés nyújtja a megoldást. Egy fejlesztőnek nagy segítséget nyújt, ha a tranzakciók lebonyolításának alacsony szintű részleteivel nem törődve, rugalmasan módosíthatja az alkalmazás tranzakciókezelését.
- **Távoli elérés.** Ha rendszerünk webes felülettel rendelkezik, az egyben egyfajta elosztottságot feltételez, hiszen az adatokat szerveroldalon tároljuk, míg a megjelenítés a klienseknél történik, böngésző segítségével. A web esetén a böngésző a HTTP protokollon keresztül kommunikál a szerverrel. Ha nem a webes, hanem a vastag kliensekre, vagy rendszerünkhöz kapcsolódó egyéb rendszerekre gondolunk, ezeknél is szükség van valamilyen protokollra, amellyel megoldható a távoli metódushívás. Ha egy olyan futási környezetre tudunk fejlesz-

teni, amely számos kommunikációs protokoll hálózati szintű részleteit elfedi a fejlesztő elől, az jelentős támogatást jelent.

- **Névszolgáltatás.** Elosztott rendszerekben elengedhetetlen követelmény, hogy bizonyos objektumokat, erőforrásokat valamilyen néven regisztrálni tudjunk, hogy azokat majd más objektumok név alapján, a konkrét hálózati cím ismerete nélkül (vagyis helyátlátszó módon) elérhessék. Ehhez a névszolgáltatás nyújt megfelelő infrastruktúrát, ezért ez is fontos eleme egy vállalati méretű alkalmazások fejlesztését támogató platformnak.

- **Skálázhatóság.** A rendszer terhelése idővel növekedhet, hiszen több bankfiók nyílhat, egyre több ügyfél veheti igénybe a webes felületet. Egy rendszert akkor nevezünk skálázhatónak, ha nagyságrendekkel megnövekedett terhelést is kezelni tud, anélkül, hogy a felhasználók jelentős teljesítménycsökkenést (pl. válaszidő-növekedést) tapasztalnának, és anélkül, hogy a programkódot módosítani kellene. A skálázás történhet **függőlegesen**, ami azt jelenti, hogy egyre erősebb hardverre (processzor, memória) telepítjük ugyanazt az alkalmazást. Ennek a skálázási módnak a felső korlátja: az ésszerű áron elérhető legerősebb hardver. Adott hardverárszint felett költséghatékonyabb megoldás a **vízszintes skálázás**, vagyis több gépből álló klaszter (együttműködő csoport) alkalmazása. Ha kezdettől fogva olyan technológiákat alkalmazunk rendszerünk fejlesztésekor, amelyek klaszterezésre alkalmasak (pl. távolról is hívható szoftverkomponensek), azzal biztosítjuk a vízszintes skálázás lehetőségét.

- **Magas rendelkezésre állás.** Legyen szó bármilyen szoftverrendszerről, a felhasználóknak kellemetlen, ha a rendszert valamilyen (akár szoftver-, akár hardver-) hiba miatt nem tudják megfelelően használni. A példánkban szereplő banki rendszer esetén az ügyfelek elégedetlensége az ügyfelek elvesztését, üzleti veszteséget jelent. Más rendszerekben emberéletek is múlhatnak egy akár csak néhány perces rendszerleálláson. A magas rendelkezésre állás biztosításának egyik lehetséges módja a klaszterezés, így ismét megállapíthatjuk, hogy igen nagy előnyt jelent, ha a futási környezetünk és az alkalmazott technológiák támogatják a klaszterezést.

- **Aszinkron üzenetkezelés.** A magas rendelkezésre álláshoz is kapcsolódik a következő probléma. Tegyük fel, hogy banki rendszerünknek egy tranzakció elvégzése miatt más bankok rendszereivel vagy valamilyen bankközi elszámoló rendszerrel kell együttműködnie. Ha ezek a partnerrendszerek éppen nem állnak rendelkezésre, amikor pl. az ügyfél átutalását szeretnénk elvégezni, akkor az ügyfél ugyanúgy hibát észlel, mint ha a mi rendszerünk állt volna le. Ez a helyzet jól kezelhető, ha egy aszinkron üzenetkezelő rendszer segítségével csak laza csatolásban állunk a partnerrendszerekkel. Ilyenkor az aszink-

ron üzenetkezelő rendszer az elküldött üzenetünket akkor is megőrzi, ha a partnerrendszer éppen nem működik. Amikor pedig újra feléled, meg fogja kapni és fel tudja dolgozni a kérésünket. Hosszan tartó műveleteket is érdemes aszinkron módon indítani, így jobb válaszidőt biztosíthatunk a felhasználóknak. Az aszinkron üzenetkezelő rendszerek a relációsadatbázis-kezelőkhöz hasonlóan igen komplex szoftverek, amelyeket nem érdemes saját magunknak megírni. Ehelyett olyan platformot kell választani a fejlesztésre, amely támogatja az elterjedt üzenetkezelő rendszerek elérését, esetleg már saját maga is tartalmaz egy aszinkron üzenetkezelőt.

- **Biztonság.** Többfelhasználós rendszereknél mindig felmerülnek biztonsági kérdések: mit tehet meg az adott felhasználó a rendszerben? (Autorizáció, azaz felhatalmazás, jogosultság problémája.) Egyáltalán minek alapján azonosítjuk a felhasználót? (Autentikáció, azaz valódiság, hitelesség problémája.) Hogyan védekezzünk a hálózati forgalom rosszindulatú módosítása vagy lehallgatása ellen? (Adatintegritás, illetve bizalmasság problémája.) Ezen kihívások igen körültekintő megoldást igényelnek, alapos kriptográfiai ismeretekkel kell rendelkezni a megfelelő algoritmusok és protokollok kiválasztásához és implementálásához. Emiatt mindenképpen érdemes saját fejlesztés helyett kész megoldásokra támaszkodni, amelyeket a futási környezet biztosít számunkra.
- **Monitorozás és beavatkozás.** Egy szoftverrendszer méretének növekedésével egyre nagyobb kihívást jelent a működés pontos nyomon követése például hibakeresés vagy statisztikakészítés céljából. Bizonyos feltételek fennállásakor pedig felmerülhet az az igény is, hogy beavatkozzunk a működésbe, megváltoztassuk alkalmazásunk konfigurációját. Nagy segítség, ha az ehhez szükséges infrastruktúrát nem a fejlesztőnek kell kialakítaniuk, hanem az futási idejű szolgáltatásként vehető igénybe.

A most felsorolt szolgáltatásokat szokás middleware-szolgáltatásoknak is nevezni, mivel rendszerint, mint a Java EE esetében is, egy köztes rendszer (middleware) biztosítja őket, vagyis az alkalmazásfejlesztők ezeket készen vehetik igénybe. A Java EE több technológiát definiál, amelyek együttesen fedik le ezeket a követelményeket. Ezen technológiák mindegyike saját API-n keresztül érhető el, egy részük már a Java standard kiadásában is. Az alábbi táblázat összefoglalja, melyek ezek a technológiák, melyik middleware-szolgáltatáshoz kapcsolhatók, és a könyv melyik további fejezetei tárgyalják részletesen.

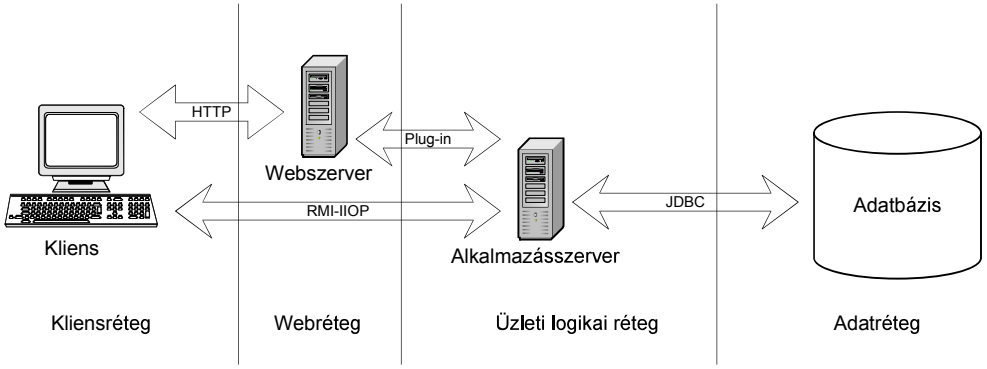
Technológia	Kapcsolódó middleware-szolgáltatás	Fejezet
Java DataBase Connectivity (JDBC)	Perzisztencia	2
Java Persistence API (JPA)	Perzisztencia	2
Enterprise JavaBeans (EJB)	Többszálúság, perzisztencia, tranzakciókezelés, távoli elérés, aszinkron üzenetkezelés, biztonság	2, 6
JavaServlet	Távoli elérés (webes), többszálúság, biztonság	3
JavaServer Pages (JSP)	Távoli elérés (webes), többszálúság, biztonság	3
JavaServer Faces (JSF)	Távoli elérés (webes), többszálúság, biztonság	4
Java Naming and Directory Interface (JNDI)	Névszolgáltatás	2
Java Message Service (JMS)	Aszinkron üzenetkezelés	6
Java Transaction API (JTA)	Tranzakciókezelés	2
Java Authentication and Authorization Service (JAAS)	Biztonság	9
SOAP with Attachments API for Java (SAAJ), Java API for XML Registries (JAXR), Java API for XML-Based RPC (JAX-RPC), Java API for XML Web Services (JAX-WS)	Távoli elérés (XML webszolgáltatások)	5
Java Connector Architecture (JCA)	Távoli elérés, többszálúság, biztonság, tranzakciókezelés, aszinkron üzenetkezelés	7
Java Management Extensions (JMX)	Monitorozás és beavatkozás	8

Vannak a fejlesztésnek olyan aspektusai is, amelyek konkrét technológiák ismeretén kívül egyéb, a gyakorlatban leginkább bevált és széles körben elterjedt ismereteket is feltételeznek. Ezek közül néhányról (biztonság, loggolás, tesztelés, más rendszerekkel történő integráció) a 9–12. fejezetekben esik szó. A 13. fejezet az internetes melléklet tartalmát ismerteti, és részlete-

sen leírja, milyen módon futtathatók a példák az ingyenesen elérhető NetBeans fejlesztőeszköz és a Sun Java System Application Server segítségével. Az egyes technológiák bemutatása után mindig utalunk rá, hogy az internetes melléklet[1] melyik példáját érdemes áttanulmányozni.

1.1. Az n rétegű architektúra és a Java EE alkalmazásszerver

Az eddigiekben gyakran hivatkoztunk a futtató környezetre, amely middledware-szolgáltatásokat nyújt az alkalmazások számára. Itt az ideje, hogy eláruljuk: Java EE esetén ez a futási környezet az ún. **alkalmazásszerver**, amely egy réteges szoftverarchitektúrába illeszkedik be az alábbi módon.



1.1. ábra *Az n rétegű architektúra*

A réteges architektúrák jellemzője, hogy minden réteg egy-egy jól definiált feladatot lát el, a közvetlenül alatta lévő réteget felhasználva, ezért fontos érteni az ábrán szereplő rétegek szerepét. Az **adatréteg** feladata az adatok perzisztens tárolása, és az azokon végezhető elemi műveletek – vagyis a létrehozás, lekérdezés, módosítás és törlés – támogatása. Leggyakrabban ezt a réteget relációs adatbázis segítségével valósítják meg, bár egyre kiforrottabb megoldást biztosítanak az objektumorientált adatbázisok is. Tágabb értelemben az adatréteghez tartozik minden rendszer, amelyből egy alkalmazás adatokat nyer ki, legyen ez akár egy mainframe, akár egy ERP-rendszer (Enterprise Resource Planning = Nagyvállalati Erőforrás-tervezés), vagy egy korábbi alkalmazás. Ilyenkor szokás az adatréteget EIS-rétegnek (Enterprise Information System = Nagyvállalati Információs Rendszer) is nevezni.

[1] A könyv előző kiadása CD-mellékletet tartalmazott. Az újabb kiadásban ezt internetes melléklettel váltottuk ki, melynek elérhetőségét a 13. fejezetben adjuk meg. Amennyiben a könyv szövegében bárhol CD-melléklet szerepel, ezen az internetes mellékletet kell érteni – a szerkesztő.

Az adatrétegre épül az **üzleti logikai réteg**, amely a konkrét alkalmazási terület (business domain) igényeinek megfelelő funkcionalitást biztosítja oly módon, hogy az üzleti szabályok figyelembevételével hívja meg az adatréteg szolgáltatásait. Egy banki alkalmazásnál például az üzleti logika feladata egy átutalás elvégzése, melynek során ellenőrzni kell a jogosultságot, majd az adatréteget felhasználva csökkenteni az egyik számla egyenlegét, és növelni a másikét. Fontos, hogy az „üzleti" szó itt nem pénzügyi értelemben jelent üzletet, egy hallgatói információs rendszerben például egy kurzus kiírása lehet egy üzleti funkció. Ez az a réteg, amelyet Java EE esetében egy alkalmazásszerverre telepítünk a middleware-szolgáltatások felhasználása érdekében.

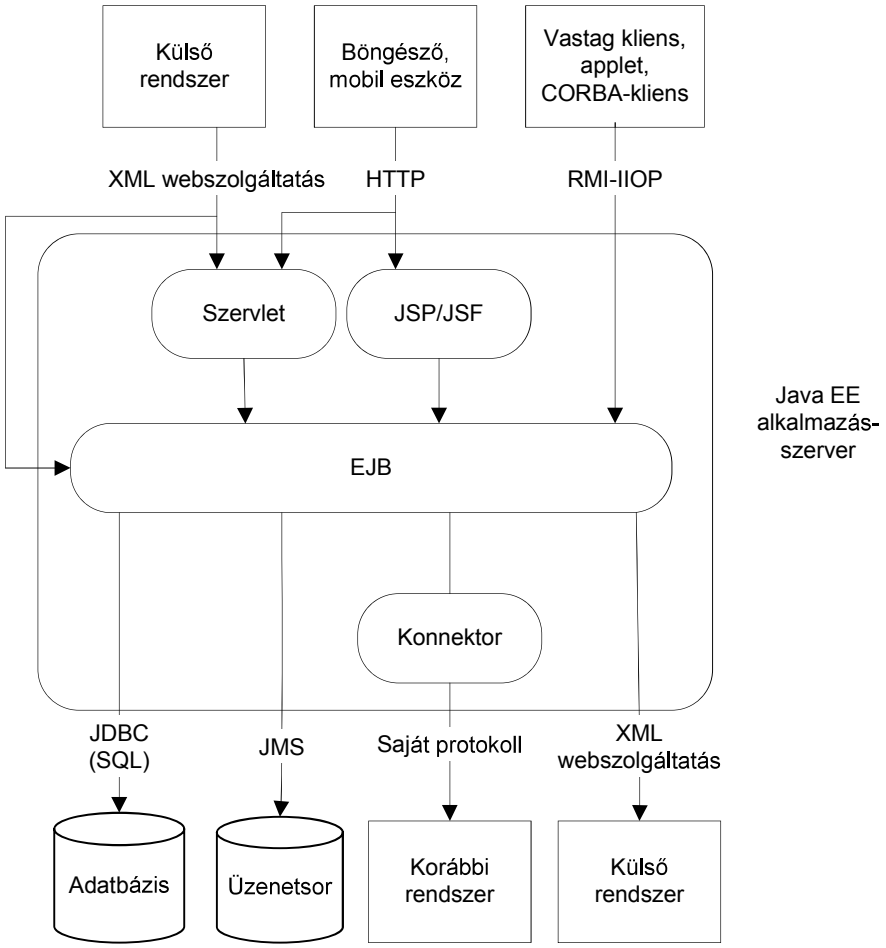
A **kliensréteg** biztosítja az alkalmazás felhasználói felületét, vagyis a felhasználói beavatkozások hatására meghívja a megfelelő üzleti logikai funkciót, majd a hívás eredményének megfelelően frissít bizonyos felhasználói felületelemeket. A kliens alapvetően két típusú lehet. Vastag kliensnek hívjuk a hagyományos desktop alkalmazásokat, vékony kliensnek pedig a böngészőt, amely webes alkalmazások esetén biztosítja a letöltött oldalak megjelenítését. Egyre inkább megfigyelhető a két klienstípus közötti határ elmosódása, az ún. gazdag webes klienseknek köszönhetően. Ezek ugyanis különféle technológiák segítségével a vastag kliensekhez hasonló felhasználói élményt biztosítanak a böngészőkön belül is. Természetesen lehetséges, hogy az alkalmazásszerveren futó üzleti logikához több klienstípus is csatlakozzon. Ilyenkor mutatkozik meg a szigorú rétegelt architektúra egyik előnye: ha az üzleti logikát valóban az alkalmazásszerverre telepített rétegbe koncentráltuk, akkor a különféle kliensekben nem kell az üzleti logikát megismételnünk, csupán a megjelenítést kell lecserélnünk.

Amennyiben vékony klienseket is ki akarunk szolgálni, szükség van egy **webréteg**re is, amelyik a böngészőktől érkező HTTP-kéréseket értelmezi, meghívja a megfelelő üzleti logikát, majd pedig megfelelő (tipikusan HTML-, de akár XML-, WML-, tetszőleges bináris formátumú) választ generál. A webréteget Java EE esetén szintén az alkalmazásszerverre telepítjük. Az alkalmazásszerver képes közvetlenül fogadni a böngészőtől érkező kéréseket, de gyakran egy webszervert is beiktatnak az alkalmazásszerver elé, amely például a statikus erőforrásokat (képek stb.) képes kiszolgálni, így egy plug-in segítségével csak azokat a kéréseket továbbítja az alkalmazásszerverhez, amelyek az üzleti logika meghívását igénylik.

A fenti architektúrára szokás 3 rétegű architektúra néven is hivatkozni, ilyenkor az adat–üzleti logika–kliensréteg felosztást értik alatta. Helyesebb azonban n rétegű architektúráról beszélni, hiszen mint láttuk, webes kliensek esetén az üzleti logika és a kliensréteg közé illeszkedik negyedikként a webréteg. Fontos látni, hogy az egyes rétegek már meglévő, készen kapható szoftverekre épülnek: az adatréteg egy adatbázis-kezelő rendszerre, az üzleti logika és a webréteg egy alkalmazásszerverre, a webréteg esetleg még egy webszerverre is, a kliensréteg pedig vékony kliens esetén egy böngészőre. Fizikailag ezek a szoftverrétegek külön gépeken is futhatnak, de közülük több

akár közös fizikai gépre is telepíthetők így a fizikai rétegek száma igen képlékeny, ezért is indokolt az n rétegű elnevezés.

Az 1.2. ábra szemlélteti, milyen szoftverkomponenseket fejleszthetünk és telepíthetünk az alkalmazásszerverre, és hogy ezek a komponensek milyen módon kommunikálnak a szerverhez kapcsolódó egyéb rendszerekkel.



1.2. ábra *A Java EE alkalmazásszerver és a hozzá kapcsolódó rendszerek*

Az üzleti logikát Enterprise JavaBeans (EJB) komponensekben szokás megvalósítani. E technológia jellemzője, hogy a fejlesztett EJB-k távolról is elérhetők, az RMI-IIOP (Remote Method Invocation over Internet Inter-ORB Protocol) protokollon keresztül anélkül, hogy a hálózati szintű részleteket a fejlesztőnek ismernie kellene. Az RMI a távoli eljáráshívás rövidítése, ez a szabványos módszer a Java világában az elosztott objektumok közötti kommunikációra, vagyis Javában írt vastag kliensek és appletek ezen keresztül érhetik el az EJB-ket. Az RMI-IIOP az RMI-t annyiban módosítja, hogy az IIOP (Internet Inter-ORB Protocol) protokollt használja a hálózati kommuni-

káció során. Miután az IIOP az elosztott rendszerek megvalósítására elterjedt CORBA-technológia (Common Object Request Broker Architecture) hálózati protokollja, az EJB-k tetszőleges programozási nyelven megírt CORBA-kliensekből is hívhatók.

A HTTP-n keresztül csatlakozó (akár mobil eszközökön futó) böngészők kiszolgálását webkomponensek végzik, amelyek szervletek, JavaServer Pages (JSP) vagy JavaServer Faces (JSF) oldalak lehetnek. A kérés kiszolgálása során ezek természetesen meghívhatják az üzleti logikát megvalósító EJB-ket. Ha egy webkomponens közös alkalmazásszerveren fut a hívott EJB-vel, akkor a hívás hagyományos Java metódushívás is lehet, ha viszont külön alkalmazásszerveren futnak, akkor a meghívás a korábban említett RMI-IIOP-n keresztül történik.

Az utóbbi években a platformfüggetlen távoli elérésre kialakult és széles körben elterjedt egy szabvány, az XML webszolgáltatások technológiája. Egy Java EE alkalmazásszerveren futó EJB, vagy akár egy webrétegbeli komponens XML webszolgáltatásokon keresztül is elérhető, és maguk a komponensek is hívhatnak más, nem feltétlenül Java-rendszerek által publikált webszolgáltatásokat.

Akármilyen protokollon keresztül érkezett is a kérés az alkalmazásszerverhez, a kiszolgálás során a komponenseknek rendszerint különféle háttérrendszerekhez kell fordulniuk. Amennyiben szigorúan betartjuk a rétegelt architektúrát, a webkomponensek csak az EJB-ket hívhatják, és csak azok érik el közvetlenül ezeket a háttérrendszereket. Lehetőség van ugyanakkor arra is, hogy a webkomponensek az EJB réteget kihagyva tegyék meg ugyanezeket a hívásokat, de az áttekinthetőség kedvéért ezt nem tüntettük fel az ábrán. Ezen háttérrendszerek közül a leggyakoribb a relációs adatbázis, amellyel a komponenseink SQL nyelven, a JDBC-technológia segítségével kommunikálnak. Aszinkron üzenetkezelés használatakor a JMS API-n keresztül fordulhatunk egy üzenetsorhoz. Elképzelhető, hogy olyan régebbi rendszerekkel kell együttműködnünk, amelyekhez csak valamilyen sajátos hálózati protokoll vagy natív hívások segítségével csatlakozhatunk. Ilyenkor a Java Connector Architecture (JCA) felhasználásával írhatunk ún. resource adapter komponenseket, amelyek az alkalmazásszerverbe telepítve úgy oldják meg a kommunikációt, hogy közben az alkalmazásszerver által nyújtott szolgáltatásokat is kihasználják.

A fent említett Java EE komponenseknek van néhány közös jellemzője. Minden komponens egy ún. konténerben fut, futási időben ez biztosítja a komponens számára a middleware-szolgáltatásokat. Az alkalmazásszervert web- és EJB-konténerre oszthatjuk, kliensoldalon applet- és vastagkliens-konténerről beszélhetünk. A Java EE komponensek lazán csatoltak, így a konkrét telepítési környezet ismerete nélkül történhet a fejlesztésük. Igen gyakran interfészeken keresztül érik el egymást, ennek előnyét a 2. fejezetben látni fogjuk. Lehetnek elosztottak, ilyenkor távolról is hívhatók, ugyanakkor helyátlátszóak, vagyis a hívónak nem kell ismernie a hívott komponens fizikai helyét. A Java EE komponensek mindig újrafelhasználható, al-

kalmazásszerverek között hordozható telepítési egységek. Mit jelent pontosan az alkalmazásszerverek közötti hordozhatóság?

A Java EE jellegzetessége, hogy nyílt szabvány, ami azzal jár, hogy több implementációja létezik. Egy implementáció gyakorlatilag egy alkalmazásszervert jelent, amely ha megfelel a Sun által előírt tesztsorozatnak, akkor megfelelő verziójú Java EE (J2EE) tanúsítványt kaphat. A tesztsorozat azt ellenőrzi, hogy az adott verziójú Java EE (J2EE) specifikációban előírt technológiáknak megfelelő API-kat (melyek rendszerint csak interfészeket definiálnak) az adott alkalmazásszerver helyesen implementálja-e. Így garantálható, hogy a specifikációval kompatibilis alkalmazásszervereken ugyanúgy futtathatók lesznek az általunk kifejlesztett, Java EE API-kat hívó komponenseink. Az elterjedt alkalmazásszerverek közé tartozik a Sun Java System Application Server, az IBM WebSphere Application Server, a BEA WebLogic Server, az Oracle Application Server, a JBoss Application Server, illetve az Apache Tomcat, amely csak webkomponensek futtatására képes, mert EJB-konténert nem tartalmaz.

Egy komplett Java EE-alkalmazás fejlesztése, telepítése, majd pedig üzemeltetése számos feladattal jár. Ezek a feladatok szerepek köré csoportosíthatók. A Komponensfejlesztő *(Application component provider)* alkalmazásszerver gyártófüggetlen komponenseket ír. Az Alkalmazás-összeállító *(Application assembler)* ezekből a komponensekből állít össze – a komponensek közötti függőségek feloldásával – egy még mindig gyártófüggetlen alkalmazást. A Telepítő *(Deployer)* telepíti ezt az alkalmazást a konkrét alkalmazásszerverre, ennek során feloldja a külső függőségeket, például integrálja az alkalmazást a meglévő biztonsági infrastruktúrába. A Rendszeradminisztrátor (System Administrator) monitorozza, és ennek alapján optimális teljesítményre hangolja a futó alkalmazást. Az Eszközgyártó *(Tool provider)* az előző tevékenységek elvégzését támogató eszközöket gyárt, végül az Alkalmazásszerver-gyártó *(Product provider, vendor)* a futási környezetet, magát az alkalmazásszervert fejleszti. Természetesen ezen szerepkörök közül akár többet is végezhet ugyanaz a személy. Könyvünk alapvetően a komponensfejlesztő, az alkalmazás-összeállító és a telepítő szerepkörökhöz szükséges ismeretekkel foglalkozik.

MÁSODIK FEJEZET

Enterprise JavaBeans

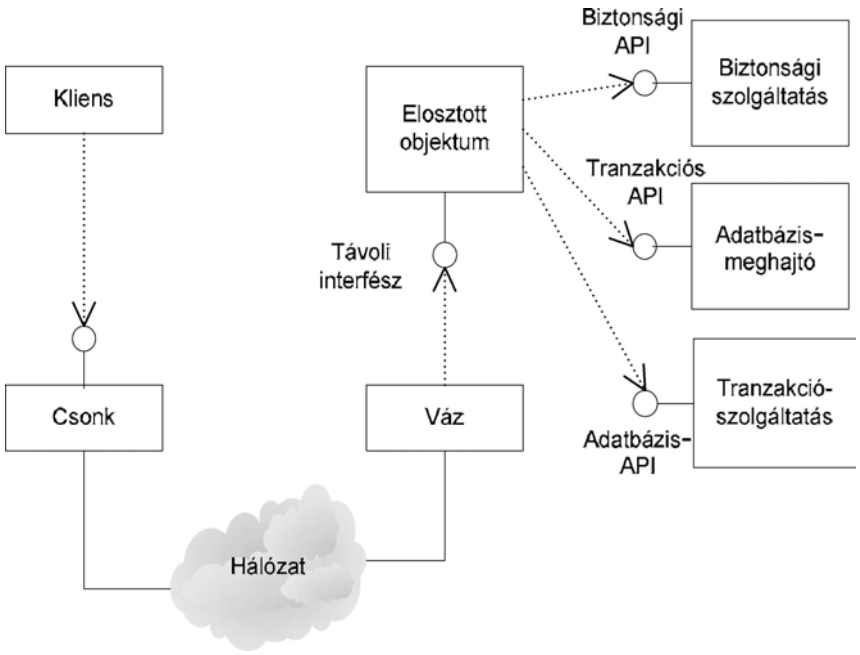
Az Enterprise JavaBeanek (**EJB**-k) a Java EE-alkalmazásokban központi szerepet betöltő, szabványos felülettel rendelkező, szolgáltatásokat nyújtó, szerveroldali komponensek, amelyeket építőelemekként felhasználhatunk egy összetett alkalmazás, illetve rendszer megalkotása során. A jelenlegi EJB-specifikáció háromfajta EJB-t definiál. A **session beanek** üzleti folyamatokat reprezentálnak (pl. hitelkártya-kezelés, árszámítás). Az egyes használati eseteknek metódusokat feleltethetünk meg, az összetartozó metódusokat pedig egy session beanhez rendelhetjük. Az **entity beanek** ezzel szemben az üzleti modellt, annak entitásait reprezentálják (pl.: hitelkártya, termék). Az entity bean a perzisztenciáért felelős adatbázissal tartja a kapcsolatot, egy példánya gyakorlatilag a relációs adatbázis egy sorának memóriabeli cacheeként értelmezhető. Végül a **message-driven beanek** az aszinkron üzenetkezelésre adnak kényelmes megoldást. Mivel ez utóbbi szorosan kapcsolódik a **Java Message Service** (**JMS**) technológiához, ezért az arról szóló fejezetben kerül tárgyalásra.

Az EJB-specifikáció legfrissebb, **3.0**-s változata ([1], [2], [3]), mely a **Java EE 5** része, jelentős újításokat tartalmaz a korábbi verziókhoz képest. Ezen újítások elsődleges célja, hogy egyszerűsítse az EJB-k fejlesztésének és használatának folyamatát, ebből következően az EJB 3.0 egészen más programozói modellt követ. Könyvünkben először az eggyel korábbi, **J2EE 1.4**-ben bevezetett **EJB 2.1**-et ([4]) ismertetjük (mely alapjaiban megegyezik a J2EE 1.3 EJB 2.0-jával), több okból. Egyrészt számos létező rendszer használja ezt a verziót, és számos jövőbeli rendszer megalkotásakor is ezt fogják választani, mert különböző szempontok miatt (anyagi okok; kezdeti, EJB 3-képes alkalmazásszerver-implementációktól való tartózkodás) sok helyen csak többéves késéssel váltanak verziót. Másrészt az EJB 2.1 ismeretében könnyebben érthető meg néhány koncepció, mely az EJB 3.0-ban kissé egyszerűsödött formában ugyan, de tovább él.

2.1. EJB 2.1

2.1.1. Egy 2.1-es EJB felépítése

Mint említettük, többrétegű architektúrák esetén a middleware-szolgáltatásokat rendszerint a középső rétegben valósítják meg, így a J2EE-rendszerekben ez a feladat éppen az EJB-kre hárul. A hagyományos módszer middleware-szolgáltatások elérésére az úgynevezett explicit middleware ([5]). (2.1. ábra)

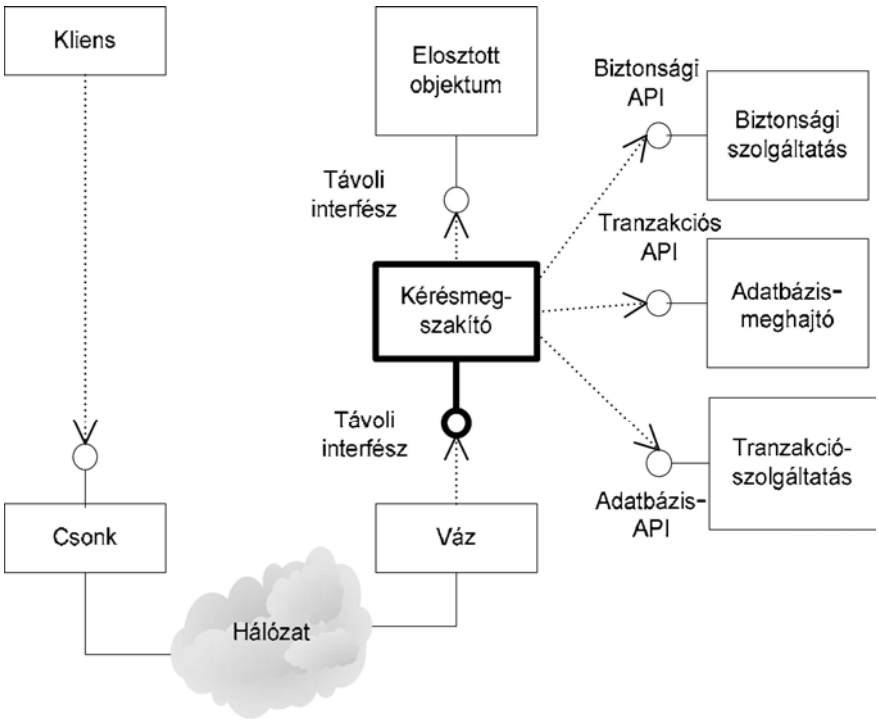


2.1. ábra *Explicit middleware*

Ebben az esetben láthatjuk, hogy a kliens kérését a **kliensoldali csonk** (stub) továbbítja a hálózaton keresztül a **szerveroldali váznak** (skeleton), amely a hálózaton keresztül elérhető, a kérést ténylegesen végrehajtó **elosztott objektum** megfelelő metódusát hívja meg. Az elosztott objektum a metódus implementációjában különféle middleware API-kat hív (pl. tranzakciós, biztonsági, adatbázis-API). Ezáltal a forráskód természetesen felduzzad, továbbá, ha a komponenst eladjuk, és a vevő másképpen akarja használni a middleware-szolgáltatásokat (például módosítani akarja a jogosultságkezelést), csak a forráskód kiadásával tudja ezt megtenni.

Ezen hátrányokat küszöböli ki az implicit middleware. (2.2. ábra) Ennek lényege, hogy egy külön leíró fájlban csak **deklaráljuk**, hogy milyen szolgáltatásokat kérünk. Így a programkódban ténylegesen csak az üzleti logikával kell foglalkoznunk, kész komponens vásárlása esetén pedig a vevő módosítani tudja a middleware-t anélkül, hogy a gyártó kiadná a komponens forráskódját. Az implicit middleware az elosztott objektumhoz érkező kérés megszakításával valósítható meg. A **kérésmegszakítót** (request interceptor) nem nekünk kell megírni, hanem a leíró fájl alapján generálódik, és annak alapján hívja a middleware API-kat.

Az explicit és implicit middleware általános, nemcsak a Java világában érvényes fogalmak. Az EJB-technológia egyik legnagyobb előnye, hogy implicit middleware-szolgáltatásokat vehetünk igénybe, amit az EJB-konténer biztosít. Ez az alkalmazásszerver kulcsfontosságú része, olyan futtatási környezetet biztosít az EJB-k számára, amely számos middleware-szolgáltatás kezelését implicit módon, vagyis fejlesztői kódolás nélkül átvállalja a programozótól.



2.2. ábra *Implicit middleware*

Ahhoz, hogy a konténer mindezt elvégezhesse, bizonyos szabályoknak megfelelően kell felépíteni az EJB-komponenst: több osztályból, interfészből és egyéb fájlokból. Ezeket ismertetjük a továbbiakban. Ezen fájlok egy részét az EJB fejlesztőjének, vagyis nekünk kell megírni. A másik felét az EJB-konténer gyártója által a konténerrel együtt szállított eszköz generálja számunkra, vagy a fejlesztési folyamat részeként, vagy telepítési, vagy futási időben. A továbbiakban generált fájl néven az ilyen fájlokra hivatkozunk, nem pedig azokra, melyeket nekünk kell létrehoznunk (még akkor sem, ha ezeket egy fejlesztőeszköz esetleg legenerálja számunkra, így nem kézzel írjuk meg).

Az első építőelem egy EJB-komponensben az **Enterprise Bean** osztály (más néven **bean osztály** vagy **implementációs osztály),** mely az üzleti logikát tartalmazza, vagyis a kliensek felé felkínált metódusok implementációit. Ez a 2.2.2. ábra elosztott objektumának felel meg. A kliensek azonban sosem érik el közvetlenül az Enterprise Bean osztályt, hanem csak egy kliensoldali csonkot, amely generált fájl.

A következő elem a szintén generált kérésmegszakító osztály, amit EJB-technológia esetén **EJB-objektumnak** hívunk. Ennek, mint már említettük, az a feladata, hogy egy leíró fájl alapján middleware API-kat hívjon, majd pedig delegálja a kérést a bean osztálynak. Az EJB-objektumot az EJB-konténer generálja számunkra. Azt azonban, hogy milyen metódusai legyenek ennek az EJB-objektumnak, le kell írnunk egy interfészben. Ezt az interfészt hívják **távoli interfésznek (remote interface)**.

Fontos kérdés, hogy miként szerez a kliens referenciát az EJB-objektumra, illetve a távoli hívás miatt az EJB-objektummal a vázon keresztül kommunikáló kliensoldali csonkra. Közvetlenül nem példányosíthatja a **new** operátorral, hiszen a kliens elől el akarjuk rejteni a generált osztályokat (amelyek konténerspecifikusak), vagyis a kliens csak azt tudja, hogy milyen interfészt kínál fel a távoli objektum. További cél a helyátlátszóság megvalósítása a referencia megszerzésekor, vagyis ne kelljen pl. IP-címet beleírni a kódba. Ilyen jellegű példányosítási problémák megoldására szolgál a **Factory method** (gyártó metódus) tervezési minta [6]. Ennek lényege: egy Factory objektumra bízzuk, hogy valamilyen interfésszel visszatérő **create()** metódusában konkrét példányokat adjon vissza. EJB-k esetében ezt a Factory szerepet a **home objektum** játssza. Ennek segítségével tud a kliens létrehozni, megszüntetni, illetve megkeresni (lásd később, entity bean esetén) olyan kliensoldali csonkokat, melyek metódusait hívva a kérés az EJB-objektumhoz, onnan pedig a bean osztályhoz fut be. Ekkor természetesen felmerül a kérdés, hogy hogyan kapunk referenciát magára a home objektumra. A megoldás ismertetése előtt be kell mutatnunk az alkalmazásszerver által nyújtott névszolgáltatást.

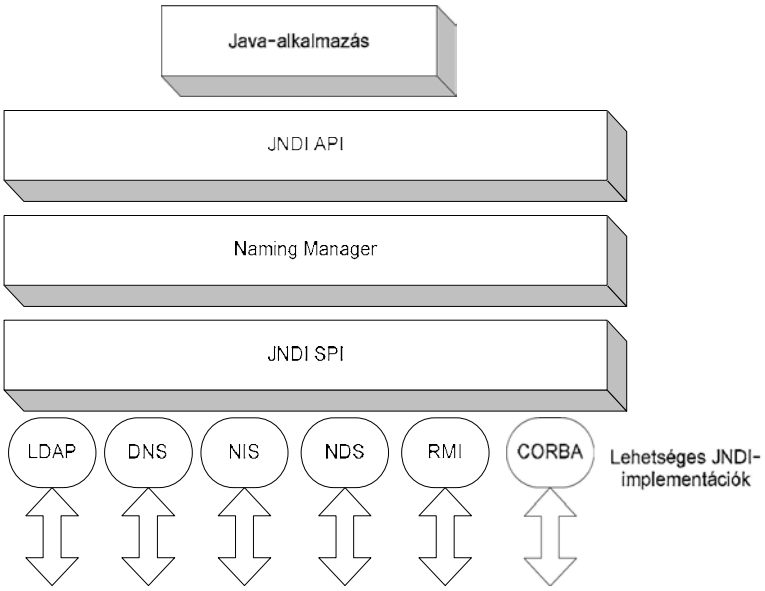
Név- és directoryszolgáltatás

Egy tetszőleges, általános névszolgáltatás objektumok és nevek összerendeléséért felelős. Egy objektum valamilyen néven regisztrálható a szolgáltatásban, majd pedig ezen a néven egy keresési művelettel elérhető. A névtér gyakran hierarchikus, pl. **java.sun.com**, **ejb/hu/bme/aut/MyEJB**. A keresés valamilyen kontextuson belül végezhető. Ha a gyökérkontextusban például **ejb/hu/bme/aut/MyEJB** néven regisztrálunk egy objektumot, akkor megtalálhatjuk a gyökérkontextusban ezen a néven keresve, vagy például az **ejb/hu/bme** kontextuson belül **aut/MyEJB** néven. A directoryszolgáltatás azzal egészíti ki a névszolgáltatást, hogy a regisztrált objektumokhoz különféle attribútumok is megadhatók, és a keresés ezek alapján is történhet. Számos implementáció létezik név- és directoryszolgáltatásra, például:

- az IP-címek és nevek összerendelését végző DNS (Domain Name System)
- a CORBA névszolgáltatása
- a Java RMI Registry
- NIS (Network Information Service)
- NDS (Novell Directory Service)
- Sun Java System Directory Server
- Windows Active Directory

Ezen szolgáltatások egy része egységesen kezelhető az LDAP (Lightweight Directory Access Protocol) interfészen keresztül.

Javában a különféle névszolgáltatásokat egységesen a **JNDI** (Java Naming and Directory Interface) API segítségével érhetjük el. Ez az API már a J2SE 1.3 óta rendelkezésre áll, az idetartozó osztályok és interfészek a **javax.naming** csomagban és alcsomagjaiban találhatók. A névszolgáltatások egységes kezelését a 2.3. ábrán látható architektúra teszi lehetővé, amely igen hasonló az adatbázisok elérését támogató **JDBC** (Java Database Connectivity) architektúrájához. Az egyes névszolgáltatásokhoz JNDI providerek (szolgáltatók, ellátók) készíthetők (a JDBC driverekhez hasonlóan), így biztosítható az egységes felület a névszolgáltatást használó alkalmazások számára. A JNDI API a Service Provider Interface-en keresztül éri el az egyes névszolgáltatás-implementációkat, vagyis egy JNDI provider elkészítéséhez a Service Provider Interface-t kell megfelelően implementálni.



2.3. ábra *A JNDI architektúrája*

A J2EE komponens modellje erőteljesen felhasználja a névszolgáltatást. Egy J2EE-alkalmazás telepítésekor a telepítő minden EJB-komponensnek ad egy JNDI-nevet, amit az alkalmazásszerveren lévő névszolgáltatás tart nyilván. Az EJB kliensei (ezek akár más EJB-k is lehetnek) pedig JNDI-név alapján keresik meg az általuk használni kívánt EJB-ket. Sőt, a külső erőforrások (pl. adatbázis, üzenetsor) elérésénél is hasonló elv érvényesül: az erőforrást valamilyen JNDI-néven regisztrálni kell az alkalmazásszerveren, a szerveren futó komponensek számára, amelyeket pedig úgy kell megírni, hogy az adott néven keressék az erőforrást. A keresés során a névfeloldás kétféle módon történhet. **Közvetlen** JNDI-nevek alkalmazásakor egyszerű a helyzet: ha például egy komponens kódja **jdbc/MyDB** néven keres egy adatbázist, akkor a névszolgáltatásban **jdbc/MyDB** néven kell regisztrálni azt. Ez a megközelítés felvet egy problémát: mi történik, ha egy másik kliens ugyanezen a néven

akar elérni egy másik adatbázist? Egy névszolgáltatásban egy névhez legfeljebb egy objektumot lehet regisztrálni, így be kell vezetni az **indirekt** JNDI-nevek fogalmát. Ennek lényege, hogy a komponens forráskódjában minden JNDI-keresés csak egy logikai névre vonatkozzon. A komponens mellé el kell készíteni egy leíró fájlt is, amelyben fel vannak sorolva ezek a logikai nevek, és hogy milyen típusú erőforrást vagy más komponenst vár az adott komponens ezen nevek alatt. Az alkalmazás összeállítása és telepítése során kell feloldani, hogy ezek a logikai nevek a tényleges telepítési környezetben milyen JNDI-nevekre képződjenek le. Így egy komponens kódjának ismerete és megváltoztatása nélkül, pusztán a logikai nevek tényleges JNDI-nevekre történő leképezésével lecserélhető az általa hivatkozott erőforrás vagy egyéb komponens. A J2EE programozói modellben emiatt indirekt JNDI-nevek használata a javasolt, amely forráskódszinten annyit jelent, hogy a keresés során a **„java:comp/env”** prefixszel kell ellátni a keresett logikai nevet.

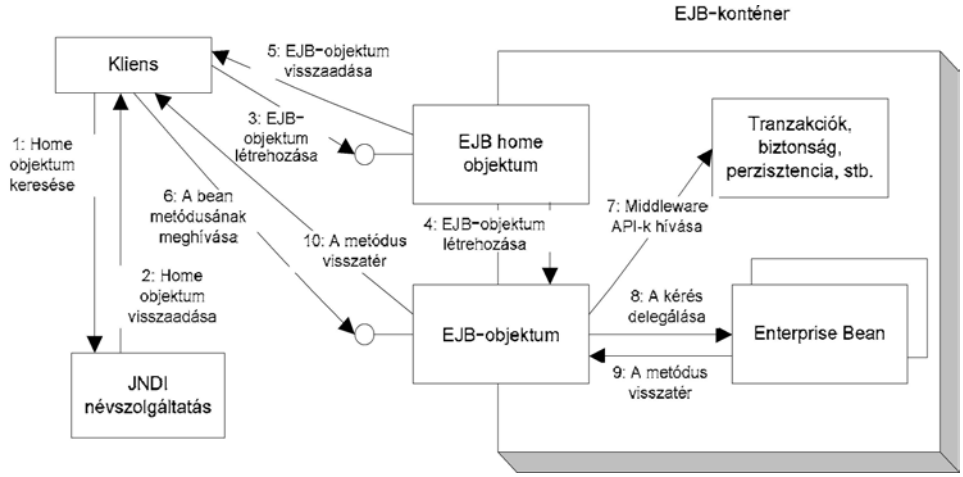
A home objektum elérése

A JNDI megismerése után visszatérhetünk eredeti kérdésünkre: hogyan szerzünk referenciát a home objektumra? A kliensek valamilyen logikai néven hivatkoznak a home objektumra, melynek feloldása telepítéskor történik meg. A home objektumot szintén az EJB-konténer generálja számunkra, és amikor egy kliens referenciát kér az EJB-objektumra, a konténer dönti el, hogy már meglévő EJB-objektumra ad-e referenciát, vagy újat hoz-e létre. Szintén a konténer dönti el, hogy egy EJB-objektum egy adott Enterprise Beanhez továbbítja-e a kliensek kéréseit, vagy többhöz – ez utóbbira a terhelés megnövekedése esetén lehet szükség. A többes esettel járó többszálú kezelést mindig a konténer végzi, az Enterprise Bean osztály írása közben tehát ezzel nem kell foglalkozni.

Visszatérve az EJB-k példányosításának problémájára, az a kérdés merült fel, hogyan szerez a kliens referenciát a home objektumra. A válasz pedig az eddigieknek megfelelően az, hogy az EJB telepítésekor egy JNDI-néven regisztrálódik az EJB home objektuma, amit a kliens – jellemzően indirekt JNDI-név alapján – meg tud keresni. A home objektummal kapcsolatban még egy nyitott kérdés maradt: ha a konténer generálja, honnan tudja, hogyan kell inicializálni az EJB-objektumot? Nos, a megoldás az, hogy egy interfészben deklaráljuk a megfelelő inicializáló metódusokat. Ezt az interfészt **home interfésznek** hívják. A home objektum ezt az interfészt implementálja úgy, hogy közben magát az inicializálást delegálja az Enterprise Bean osztálynak. Vagyis az Enterprise Bean osztály inicializáló metódusait nem a távoli interfészben kell deklarálni, mint a többi metódusát, hanem a home interfészben. Egy EJB meghívásának teljes folyamatát a jobb érthetőség kedvéért a 2.4. ábrán foglaljuk össze ([5]).

Felmerülhet a kérdés, hány példány létezhet egy EJB home objektumából, EJB-objektumából és implementációs osztályából. Ez általában konténerfüggő, de a fejlesztőt ez általában nem is érinti. Egy gyakori megoldás, hogy egy home

objektum jön létre egy EJB-típushoz, amely kezeli a konkurenciát, és több kliens kérésére akár több EJB-objektumot hozhat létre. Implementációs osztályból jellemzően több van, a konténer készletezi őket (instance pool = példánykészlet), és garantálja, hogy ebből egy példányt egyszerre csak egy szál használ, így a fejlesztőnek nem kell szinkronizációs kérdésekkel foglalkozni. Az EJB-objektum lehet 1–1 vagy 1–n kapcsolatban az implementációs osztállyal.



2.4. ábra *A kliens együttműködése az EJB-komponenssel*

Mikor az EJB-t a home interfészen, illetve távoli interfészen keresztül érjük el, valójában távoli eljáráshívások történnek, melyek jelentős overheadet képviselnek a paraméterek szerializálása (sorosítása) és deszerializálása miatt. Ha a hívó és az EJB egy szerveren fut, ez teljesen fölösleges, ezért az EJB 2.0-ban bevezették a **lokális interfész** és **a lokális home interfész** fogalmát. Ezek kiküszöbölik ezt az overheadet, viszont ily módon természetesen csak az EJB-kel azonos Java virtuális gépen futó kliensek érhetik el az EJB-t. Továbbá lokális interfészek alkalmazásakor a paraméterátadás referencia szerint fog történni, míg távoli interfész esetén ez érték szerinti volt, vagyis a hívások szemantikája is változik. Minden EJB-komponens a távoli és a lokális interfészek közül legalább az egyiket tartalmazza.

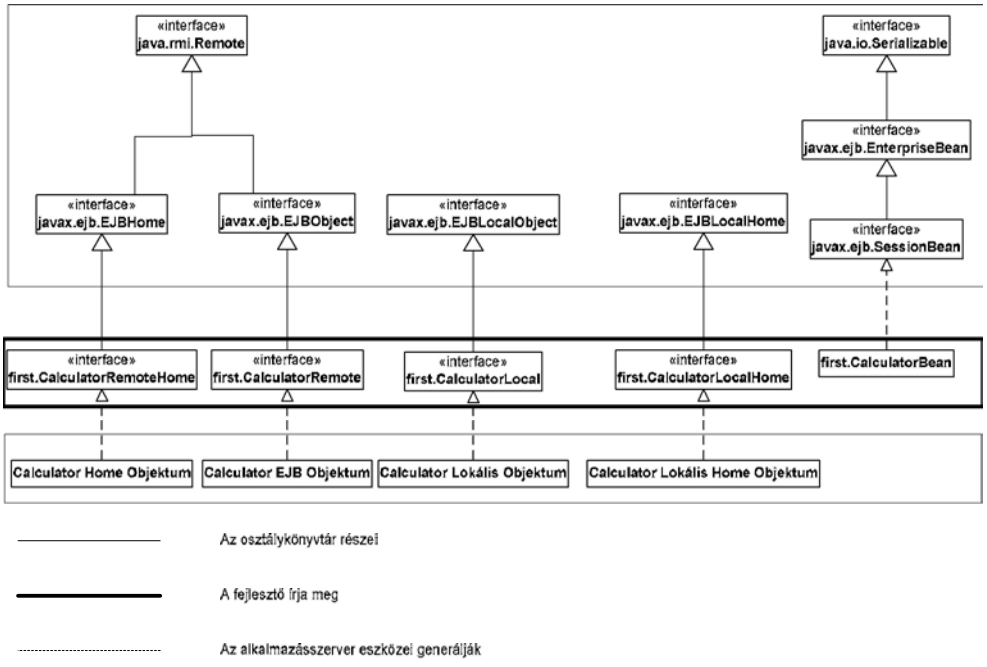
Többször volt már szó arról a leíró fájlról, amelyben a kívánt middleware-szolgáltatásokat deklarálhatjuk. Ez egy XML-formátumú fájl, az ún. **telepítésleíró** (**deployment descriptor**), amely minden EJB-komponens kötelező alkotóeleme. A middleware-szolgáltatások deklarálásán kívül itt kell leírni a komponens által tartalmazott EJB-ket, és azt, hogy ezek milyen külső erőforrásokra, más EJB-kre hivatkoznak. Minden EJB-komponens tartalmaz **gyártóspecifikus fájlokat** is, ezekben adhatók meg olyan extra szolgáltatások beállításai, melyek nincsenek benne az EJB-specifikációban, de melyeket az egyes J2EE-implementációk felkínálhatnak.

Egy EJB-komponens fizikailag egy .jar fájlként testesül meg, amelyben összetömörítve találhatók az eddig bemutatott alkotóelemek:

- Enterprise Bean osztály (implementációs, bean osztály)
- EJB-objektum
- Home objektum
- Távoli interfész (kimaradhat, ha van lokális)
- Home interfész (kimaradhat, ha van lokális)
- Lokális interfész (kimaradhat, ha van távoli)
- Lokális home interfész (kimaradhat, ha van home interfész)
- Telepítésleíró (kötelezően a **META-INF/ejb-jar.xml** fájl)
- Gyártóspecifikus fájlok

2.1.1.1. Egy egyszerű EJB

Ebben a szakaszban egy egyszerű EJB és az azt hívó kliens forráskódját ismertetjük az eddigiek illusztrálására. A **Calculator** EJB a legegyszerűbb típusú, ún. állapotmentes (lásd később) session bean, amely két szám összeadására képes. A komponenst felépítő fájlok öröklési hierarchiáját az alábbi osztálydiagram szemlélteti. Fontos kiemelni, hogy csak az alulról második sorban lévő interfészeket és osztályokat kell a fejlesztőnek megírni, a legalsó sorban konténer által generált osztályok találhatók.



2.5. ábra *A Calculator EJB osztálydiagramja*

A távoli interfész: CalculatorRemote.java

Ebben deklaráljuk az egyetlen metódust, ami a kliensek által hívható lesz. Az **add()** metódus, mint minden távolról hívható metódus, **java.rmi.RemoteException**-t dobhat, amit kötelező feltüntetni. A konténer által generált EJB-objektum fogja implementálni ezt az interfészt, amely a javax.ejb.EJBObject-et terjeszti ki, s olyan metódusokat tartalmaz, amelyek minden EJB-re meghívhatók. Ilyen például a **remove()** metódus, amellyel az EJB kliensei jelezhetik, hogy nem használják tovább az objektumot.

```
package first;

public interface CalculatorRemote extends javax.ejb.EJBObject {
    double add(double a, double b) throws java.rmi.RemoteException;
}
```

A lokális interfész: CalculatorLocal.java

Ezen interfész megírásának köszönhetően az EJB-vel egy JVM-ben lévő kliens lokális hívásokkal is elérheti majd a Calculator EJB-t. A szembetűnő különbség ennek megfelelően a távoli interfészhez képest az, hogy az **EJBLocalObject** interfészt terjeszti ki, és metódusai nem deklarálják a **java.rmi.RemoteException**-t.

```
package first;

public interface CalculatorLocal extends javax.ejb.EJBLocalObject {
    double add(double a, double b);
}
```

A home interfész: CalculatorRemoteHome.java

A home interfészben deklarálunk egy **create()** metódust, ami nemcsak **RemoteException**-t, hanem **javax.ejb.CreateException**-t is dobhat.

```
package first;

public interface CalculatorRemoteHome extends javax.ejb.EJBHome {
    first.CalculatorRemote create()  throws
     javax.ejb.CreateException, java.rmi.RemoteException;
}
```

A lokális home interfész: CalculatorLocalHome.java

A távoli home interfésztől annyiban különbözik, hogy a **javax.ejb.EJBLocalHome** interfészből származik, és a **create()** metódus – mivel nem távoli hívás – nem dobhat **RemoteException**-t, és a lokális interfésszel tér vissza.

```
package first;

public interface CalculatorLocalHome extends javax.ejb.EJBLocalHome
{
    first.CalculatorLocal create()  throws
     javax.ejb.CreateException;
}
```

Az implementációs osztály: CalculatorBean.java

Végül lássuk a bean osztályt, amelyben implementáljuk a kliensek felé nyújtott szolgáltatásokat (ezek az ún. üzleti metódusok), továbbá mivel az osztályunk implementálja a **javax.ejb.SessionBean** interfészt, egyéb metódusokat is tartalmaz. Néhány megjegyzés a kóddal kapcsolatban:

- Az **ejbCreate()** metódust azután hívja meg a konténer, miután új példányt hozott létre a bean osztályból. Itt kell elvégeznünk a bean inicializálását. Ne feledjük, hogy a kliensek sosem közvetlenül példányosítják az EJB-t. Ha egy kliens a home interfész **create()** metódusát meghívja, a konténer dönthet úgy, hogy új példányt hoz létre, ekkor a kliens oldali **create()** hívást a konténer általi **ejbCreate()** hívás követi. Ez azonban egyáltalán nem szükségszerű. A konténer újrahasznosíthat egy meglévő bean példányt is, ekkor nem történik újabb **ejbCreate()** hívás. Ez természetesen csak akkor lehetséges, ha a kliensek nem tárolnak állapotot a szerveroldali objektumpéldányokban. Az állapotkezelést később részletezzük.
- Az **ejbRemove()** metódusban szabadíthatja fel a bean az általa használt erőforrásokat. Ezt szintén a konténer hívja meg, ha úgy dönt, hogy megszüntet egy bean példányt, ami nem feltétlenül esik egybe a kliens **remove()** hívásával.
- Az **ejbActivate()** és **ejbPassivate() metódusoknak csak állapottal rendelkező session bean esetén van jelentőségük, ezért ezekre ott térünk vissza.**
- A **setSessionContext()** metódust a konténer az **ejbCreate()** előtt hívja meg, és paraméterként átadja a beanhez tartozó ún. **SessionContext**-et. Implementációja gyakorlatilag mindig az ittenihez hasonló, vagyis egy példányváltozóban eltároljuk a megkapott értéket. A **SessionContext** egyfajta gatewayként képzelhető el az EJB-konténer felé. Ez tárol olyan információkat, amiket a konténer tart nyilván, és amire a beannek működése közben szüksége lehet (pl.: melyik felhasználótól érkezett éppen a hívás). Felhasználásának lehetőségeire a későbbi szakaszokban visszatérünk.

```
package first;

import javax.ejb.*;

public class CalculatorBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext context;

    //
    // A SessionBean interfész által megkövetelt metódusok
    //
    public void ejbCreate() {
        System.out.println("ejbCreate()");
    }

    public void ejbRemove() {
        System.out.println("ejbRemove()");
    }

    public void ejbActivate() {
        System.out.println("ejbActivate()");
    }

    public void ejbPassivate() {
        System.out.println("ejbPassivate()");
    }

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
        System.out.println("setSessionContext()");
    }

    //
    // Saját, tényleges üzleti metódusok
    //
    public double add(double a, double b) {
        return a+b;
    }
}
```

A telepítésleíró: ejb-jar.xml

Ebben először leírjuk, hogy ez a komponens tartalmaz egy session beant, adunk neki egy nevet (**ejb-name**), amivel később hivatkozhatunk rá. A **display-name** elemben lévő érték a fejlesztés vagy telepítés során felhasznált grafikus eszközök számára nyújt információt. Ezt követi a beant alkotó interfészek és osztályok felsorolása. Végül leírjuk, hogy állapotmentes a beanünk, és hogy a konténer felelős a tranzakciókezelésért. Ha több EJB-t csomagolnánk egybe a Calculator EJB-vel, azok mindegyike egy-egy hasonló elemként jelentkezne a telepítésleíró **enterprise-beans** elemén belül.

```
   <?xml version="1.0" encoding="UTF-8"?>
   <ejb-jar version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
     <display-name>Labor1</display-name>
     <enterprise-beans>
       <session>
         <display-name>CalculatorSB</display-name>
         <ejb-name>CalculatorBean</ejb-name>
         <home>first.CalculatorRemoteHome</home>
         <remote>first.CalculatorRemote</remote>
         <local-home>first.CalculatorLocalHome</local-home>
         <local>first.CalculatorLocal</local>
         <ejb-class>first.CalculatorBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
       </session>
     </enterprise-beans>
   </ejb-jar>
```

Gyártóspecifikus fájl: sun-ejb-jar.xml

Itt a **Sun Java System Application Server** által alkalmazott formátumban egyetlen dolgot definiálunk: telepítési időben milyen JNDI-nevet kapjon a Calculator EJB.

```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
Application Server 8.1 EJB 2.1//EN"
"http://www.sun.com/software/appserver/dtds/sun-ejb-jar_2_1-1.dtd">
   <sun-ejb-jar>
     <enterprise-beans>
       <ejb>
         <ejb-name>CalculatorBean</ejb-name>
         <jndi-name>ejb/CalculatorBean</jndi-name>
       </ejb>
     </enterprise-beans>
   </sun-ejb-jar>
```

A bean használata vastag kliensből: CalculatorClient.java

A bean használatának első lépéseként megkeressük a home objektumot. A **javax.naming.Context** objektum reprezentálja a névszolgáltatás gyökerét. Ennek a **lookup()** metódusában egy sztringet adunk meg, ami az egyszerűség kedvéért itt közvetlen JNDI-név. A keresés eredményeként kapott home objektumot megfelelő típusra kell konvertálnunk. Utána a **create()** metódus meghívásával referenciát kapunk az EJB-objektumra (pontosabban annak kliensoldali csonkjára, amely implementálja az EJB távoli interfészét), amelynek már meghívhatjuk a metódusait.

```
package first;

import javax.naming.Context;
import javax.naming.InitialContext;

public class CalculatorClient {

   public static void main(String[] args) throws Exception {

      //a névszolgáltatás inicializálása
      Context ctx = new InitialContext();

      //a home objektum keresése, általános Object típust kapunk
      Object obj = ctx.lookup("ejb/CalculatorBean");

      //a keresési eredmény konvertálása, kicsit összetett,
      //a CORBA interoperabilitás miatt
      CalculatorRemoteHome home = (CalculatorRemoteHome)
            javax.rmi.PortableRemoteObject.narrow(
                   obj, CalculatorRemoteHome.class);

      //így kapunk referenciát az EJB-objektumra,illetve a csonkjára
      CalculatorRemote calculator = home.create();

      //meghívjuk a metódusát
      System.out.println(calculator.add(1, 2));

      //majd eltávolítjuk
      calculator.remove();
   }
}
```

Ha lokális interfészen keresztül hívnánk az EJB-t, a kód így módosulna:

```
       //a keresési eredmény konvertálása egyszerűsödik
      CalculatorLocalHome home = (CalculatorLocalHome) obj;

      //így kapunk referenciát az EJB-objektumra
      CalculatorLocal calculator = home.create();

      //meghívjuk a metódusát
      System.out.println(calculator.add(1, 2));
```

Feltűnhet a 2.5. ábra osztálydiagramján, hogy az implementációs osztály (**CalculatorBean**) nem implementálja sem a távoli, sem a lokális interfészt (**CalculatorRemote, CalculatorLocal**). Ugyanakkor, mivel az interfészeket implementáló, konténer által generált EJB-objektum delegálni fogja a metódushívásokat az implementációs osztálynak, nyilvánvalóan szükséges, hogy az interfészekben deklarált üzleti metódusoknak megfelelő metódusimplementációk jelen legyenek az implementációs osztályban. Ezzel kapcsolatban két kérdés merülhet fel:

- Miért nem implementálja az implementációs osztály a saját távoli/lokális interfészét az **implements** kulcsszóval?
- Hogyan érhető el, hogy mégis kötelező legyen a megfelelő metódusok megírása az implementációs osztályban?

A válasz a következő: egyrészt szintaktikailag nincs akadálya, hogy a CalculatorBean osztály deklarációját módosítsuk:

```
   public class CalculatorBean implements javax.ejb.SessionBean,
CalculatorRemote{
```

Ekkor a fordító kikényszerítené az üzleti metódusok implementálását, vagyis a második kérdés fel sem merülne, ugyanakkor ez a megoldás két negatív következménnyel járna: mivel a **CalculatorRemote** kiterjeszti az **EJBObject** interfészt, előír olyan metódusokat is, amelyeknek csak a generált EJB-objektumban van értelme (pl. **getEJBHome()**), így ezekre valamilyen üres implementációt kellene írni az implementációs osztályban. A másik probléma ismertetéséhez képzeljünk el egy olyan metódust, amelyiknek EJB a visszatérési értéke. Mivel a kliensek csak a távoli/lokális interfészen érhetik el az EJB-t, ez így nézne ki:

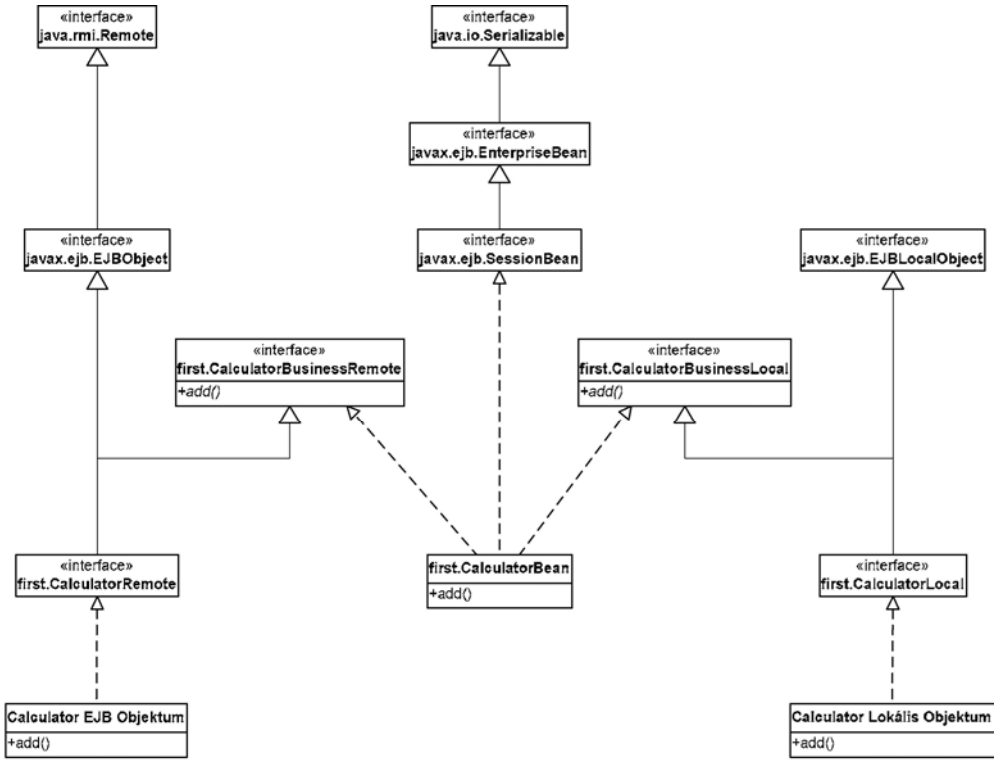
```
   public CalculatorRemote doSomething(){…}
```

Ebben a metódusban olyan objektumot kell visszaadni, amelynek metódusait a kliens hívhatja, vagyis az EJB-objektumot (vagy a vele kommunikáló csonkot), és semmiképpen nem szabad visszaadni egy **CalculatorBean** példányt. Ez utóbbi esetben ugyanis az EJB-objektum megkerülése miatt nem kerülnének meghívásra a deklarált middleware-szolgáltatások. Ha a **CalculatorBean** viszont implementálná a saját interfészét, semmi sem gátolná meg a fejlesztőt, hogy **this** referenciával, vagyis egy **CalculatorBean** példánnyal térjen vissza. Emiatt ellenjavallt a távoli/lokális interfészek implementálása.

Ezek után térjünk rá a második kérdésre. A válasz igen egyszerű: minden EJB-konténer gyártója a konténer mellé szolgáltat egy validátort. Ez képes egy lefordított, összetömörített jar-fájlt ellenőrizni, hogy megfelel-e bizonyos követelményeknek. Ezek egyike, hogy az implementációs osztályban jelen vannak-e az interfészben deklarált metódusok, vagy például hogy a telepítésleíróban megadott osztályok, interfészek megtalálhatók-e a jar-fájlban. Ezt a fejlesztési vagy legkésőbb a telepítési folyamat során le kell futtatni, így csak működőképes EJB-modult lehet telepíteni az alkalmazásszerverre.

Ha valaki mégis makacsul ragaszkodna ahhoz, hogy az implementációs osztály és a lokális/távoli interfészek közötti inkonzisztencia még fordítási időben kiderüljön, annak a **Business Interface** nevű EJB tervezési minta nyújt megoldást (2.6. ábra). Ennek lényege, hogy az üzleti metódusokat külön interfészben (business interfész) deklaráljuk, a távoli/lokális interfész pedig ezt terjeszti ki. Mivel lokális interfészek esetén a metódusoknak nem kell

RemoteException-t dobniuk, lokális esetben egy ettől különböző interfészre van szükség. Az implementációs osztály ebben az esetben sem a lokális/távoli interfészt fogja implementálni, hanem a business interfészeket. Így mindkét korábban említett probléma megoldódik: mind a távoli/lokális interfésszel visszatérő metódusban a **this** referencia visszaadása esetén, mind pedig az üzleti metódusok nem megfelelő implementációja esetén fordítási hibát kapunk. A megoldás egyetlen hátránya, hogy egy (vagy két) újabb interfészt kell karbantartani, növelve ezzel az EJB-komponensben lévő fájlok számát.



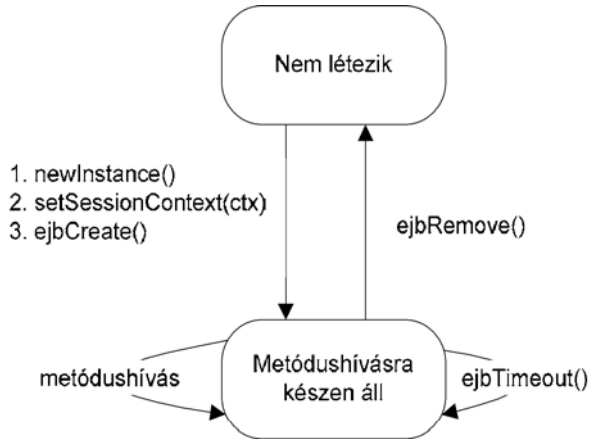
2.6. ábra *A Business Interface tervezési minta*

2.1.2. A session bean

A következő szakaszokban az EJB-k egyes típusainak jellegzetességeit fogjuk megvizsgálni. Mint már említettük, a session beanek jelentik a legegyszerűbb EJB-típust. Üzleti folyamatokat reprezentálnak, metódusaik alkotják a szerveralkalmazás által nyújtott szolgáltatások összességét. Egy banki példával élve, egy **Bank** session beannek például **createAccount()**, **deposit()**, **withdraw()**, **transfer()** metódusai lehetnének. Egy könyvtári alkalmazásban a **Library** session bean **addBook()**, **addReader()**, **borrow()** metódusokkal rendelkezhetne.

Egy session bean rendelkezhet állapottal, vagy lehet állapotmentes. Az állapotmentesség azt jelenti, hogy a kliens nem számíthat arra, hogy végig

ugyanazzal a beannel fog kommunikálni, emiatt nem tárolhat benne állapotot. A telepítésleíró **<session-type>** elemében kell deklarálnunk, hogy a session bean melyik altípusba tartozik. Az állapotmentes bean (mint az előző szakaszban megvalósított Calculator EJB) jellemzője, hogy az inicializáló **ejbCreate()** metódusának nincs bemenő paramétere. Ez szükségszerű, hiszen egy kliens hiába akarná paraméterekkel inicializálni az objektum állapotát, a következő kérése már lehet, hogy más példányhoz futna be. Ennek megfelelően az állapotmentes session beannek igen egyszerű az életciklusa (2.7. ábra). Minden bean példányt a konténer hoz létre – nem feltétlenül a kliens **create()** hívásának hatására – majd meghívja rajtuk a **setSessionContext()** és **ejbCreate()** metódusokat. Innentől bármelyik példányhoz bármelyik kliens által kezdeményezett hívás befuthat az EJB-objektumon keresztül. Időzített hívások esetén (lásd később), a konténer hívja meg rajtuk az **ejbTimeout()** metódust. Végül, mikor a konténer úgy dönt, bármely példányt megszüntethet, előtte pedig meghívja az **ejbRemove()** metódust.

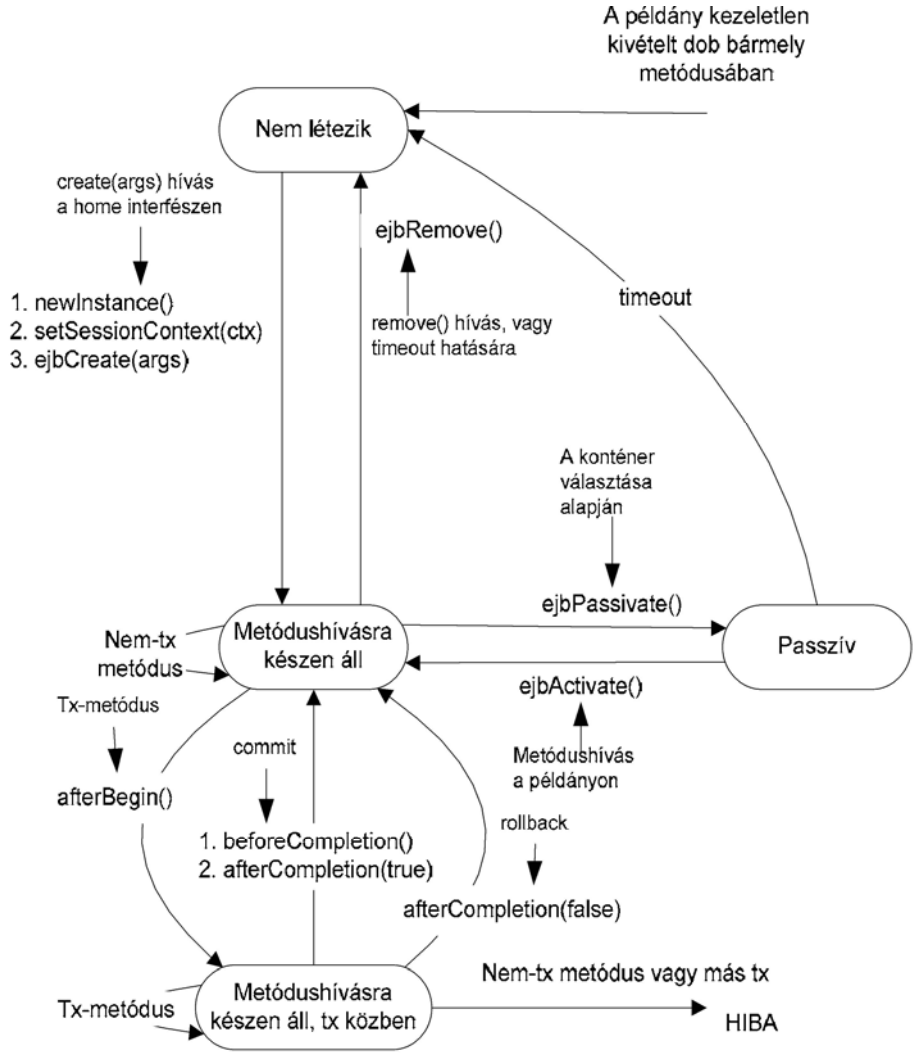


2.7. ábra *Az állapotmentes session bean életciklusa (forrás: Enterprise JavaBeansTM Specification, Version 2.1)*

Mikor arra van szükségünk, hogy egy kliens több híváson keresztül állapotot tároljon a session beanben, akkor használjuk az állapottal rendelkező session beant. Tipikus példa erre egy online bevásárló kocsi. Rendelhetnénk minden egyes klienshez külön bean példányt. Ezzel az a probléma, hogy előbb-utóbb elfogy a memória, ekkor szükség van arra, hogy meglévő beaneket elvegyünk régebbi kliensektől, és új klienshez rendeljük őket. Ennek során a bean állapotát valamilyen háttértárra menti a konténer, ezt hívják passziválásnak. Mikor ismét a régi klienst szolgáljuk ki, az állapotot vissza kell tölteni valamelyik bean példányba, amelyet előtte persze szintén passziválni kell, ha még kapcsolatban áll egy klienssel. Ez a fordított folyamat az aktiválás. Hogy mindez megvalósulhasson, a bean osztály minden elmenteni kívánt példányváltozójának szerializálhatónak kell lennie, illetve az elmenteni nem kívánt változókat a **transient** kulcsszóval kell megjelölni. A konténer dönti el, mikor

passzivál egy beant, egy szabályt betartva: kötelező memóriában tartani a beant, ha épp egy metódust hajt végre, vagy ha érintett egy tranzakcióban.

Egy bean tarthat olyan erőforrásokat, amiket nincs értelme elmenteni. Ilyenek a socketek, adatbázis-kapcsolatok. Passziválás előtt ezeket fel kell szabadítani, ezt az **ejbPassivate()** metódusban kell megtenni, amit a konténer meghív passziválás előtt. Aktiválás után pedig természetesen újra kell igényelni ezekhez az erőforrásokat, az **ejbActivate()** metódusban, amelyet szintén a konténer hív meg. Fontos megjegyezni, hogy nem a mi feladatunk az állapot mentése, ezt a konténer végzi, nekünk az **ejbPassivate()**-ben csak a bean által tartott erőforrásokat kell elengedni. Az állapottal rendelkező session bean életciklusát a 2.8. ábra foglalja össze.

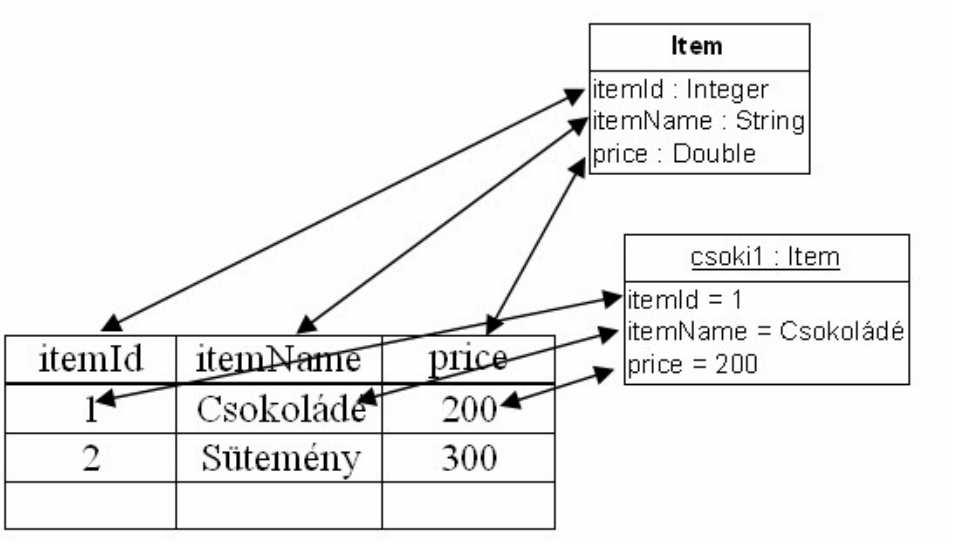


2.8. ábra *Az állapottal rendelkező session bean életciklusa (forrás: Enterprise JavaBeansTM Specification, Version 2.1)*

Az első különbség az állapotmentes esethez képest, hogy több különféle paraméterezésű **ejbCreate()** metódus is definiálható, melyek mindegyikéhez megfelelő **create()** metódus tartozik a home/lokális home interfészben. Itt a kliens **create()** utasítását követően hívódik meg az **ejbCreate()** metódus, hiszen az egyes beanek kliensspecifikus állapotokat tárolnak. A tranzakciókezeléssel (tx) kapcsolatos állapotátmenetekkel (az ábra alsó állapota) egy későbbi szakaszban (2.1.4.) foglalkozunk részletesen. Gyakorlati szempontból igen fontos a passzív állapotból a nem létezőbe vezető „timeout" átmenet, amit a régóta nem hivatkozott állapotban lévő példányokon hajt végre. Ez nem azonos az állapotmentes session bean **ejbTimeout()**-jával, hiszen az egy alkalmazás által szándékosan beállított időzítés lejártakor hívódik meg. Látható, hogy timeout esetén a konténer nem hívja meg a bean **ejbRemove()** metódusát. A konténer esetleges összeomlásakor sem történik ez meg. Emiatt, ha az **ejbRemove()** metódusban például a bean példány adatbázisműveletet végezne, érdemes egy kiegészítő programot készíteni, amely elvégzi ezt azon bean példányok helyett, amelyek nem kaptak erre lehetőséget.

2.1.3. Az entity bean

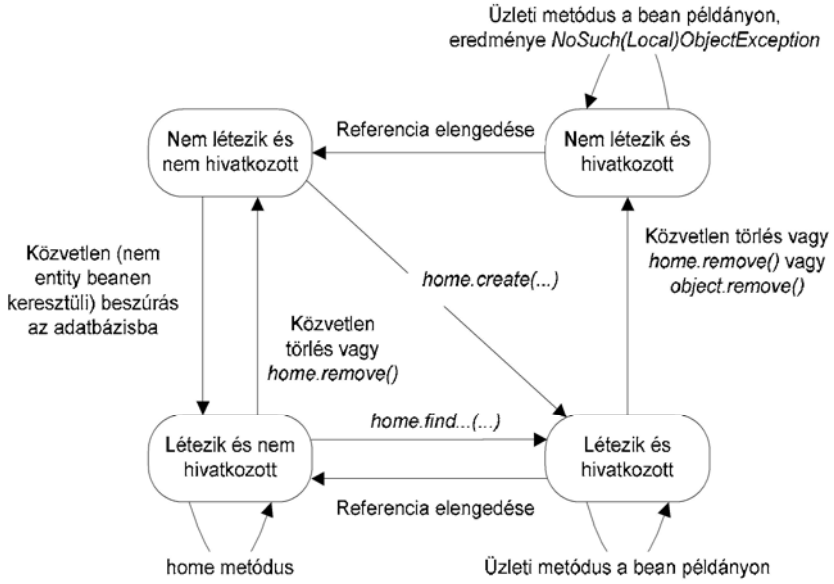
Egy entity bean feladata, hogy az általa reprezentált objektum adatait perzisztens módon eltárolja, és azok elérését biztosítsa. Mivel az alkalmazások döntő többsége relációs adatbázist használ e célra, az entity beanek oldják meg az ún. objektumrelációs leképezést (ORM – Object Relational Mapping) (2.9. ábra). Ennek lényege, hogy a relációs adatbázis tábláit osztályoknak (item), a táblák oszlopait az osztályok attribútumainak felelteti meg. Így egy adott osztály példányai (pl. csoki1) a megfelelő tábla egyes sorainak memóriabeli gyorsítótárának tekinthetők.



2.9. ábra *Az objektumrelációs leképezés*

Akárcsak egy adatbázisbeli tábla, az entity bean is rendelkezik elsődleges kulccsal, mely egyértelműen azonosítja az entity bean példányt. Az objektumrelációs leképezés megvalósítása érdekében az entity bean inicializáló metódusában (**ejbCreate()**) SQL **INSERT** műveletet, míg megszüntető metódusában (**ejbRemove()**) SQL **DELETE** műveletet kell végrehajtani. A memóriabeli bean példány és az adatbázis közötti adatszinkronizációt az **ejbLoad()** és **ejbStore()** nevű metódusok oldják meg, melyeket minden entity beannek implementálnia kell. Ezek a metódusok ennek megfelelően SQL **SELECT,** illetve SQL **UPDATE** utasításokat tartalmaznak, és jellegzetességük, hogy a konténer hívja meg őket, vagyis tulajdonképpen nem nekünk kell törődni az adatszinkronizáció feladatával: egy adott bean példány valamilyen attribútumának megváltoztatása, azaz például a **price** attribútum esetén a **setPrice()** metódusának meghívása után a konténer gondoskodik róla, hogy a változást perzisztenssé tévő **ejbStore()** metódus meghívódjon. Az entity beanek home objektuma a session beanekéhez képest új típusú metódusokat tartalmaz, amelyekkel tetszőleges paraméterek alapján megkereshetők konkrét entity bean példányok. Ezek az úgynevezett **finder** metódusok egy példánnyal, vagy példányok egész kollekciójával térnek vissza, és implementációjukban SQL **SELECT** műveletek találhatók. A finder metódusok visszatérési értéke egy találat esetén ezt az egyetlen találatot reprezentáló entity bean példány elérését biztosító csonk (ilyenkor a finder visszatérési típusa az entity bean távoli vagy lokális interfésze), több találat esetén pedig egy olyan **java.util.Collection**, amelyben az entity bean távoli vagy lokális interfészeit megvalósító objektumok találhatók.

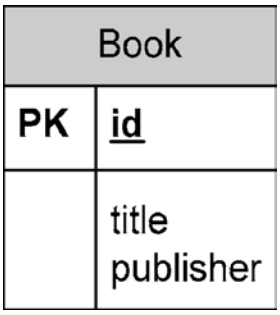
Ezen metódusok szerepét egy entity bean életciklusa során a 2.10. ábra szemlélteti.



2.10. ábra *Az entity bean életciklusa a kliens szempontjából (forrás: Enterprise JavaBeansTM Specification, Version 2.1)*

Az ábra megkülönbözteti a **létező (exists)** és a **hivatkozott (referenced)** entity bean fogalmát. Egy bean példány akkor létezik, ha a neki megfelelő adatbázissor létezik, míg hivatkozottnak akkor mondjuk, ha az entity bean kliense egy referenciát tart a memóriabeli beanre, távoli vagy lokális interfészen keresztül. Ennek megfelelően, ha egy tetszőleges, entity beanektől független alkalmazás beszúr egy sort egy táblába (direct insert, az ábra bal oldalán), akkor az e sort reprezentáló entity bean példány létező, de nem hivatkozott állapotba kerül. Ha viszont az entity beant a home interfésze **create()** metódusának valamely, a bean fejlesztője által túlterhelt változatával hozzák létre, akkor nem csak létező, de hivatkozott is lesz. Ebben az állapotban a bean példány tetszőleges üzleti metódusa meghívható. Az adatbázisban már létező beanre referencia a home interfész finder metódusaival szerezhető. Ha a beanre mutató összes referencia élettartama megszűnik, a bean újra nem hivatkozott állapotba kerül. Egy bean többféleképpen vihető nem létező állapotba: a rá szerzett referencia **remove()** metódusát meghívva, vagy a home interfész **remove()** metódusát meghívva (ekkor az elsődleges kulcsát meg kell adni bemenő argumentumként), vagy az entity beanektől függetlenül, egy közvetlen adatbázisbeli törléssel (direct delete). Ha bármelyik típusú törlés után referencia marad egy nem létező entity beanre, akkor azon keresztül bármilyen metódust hívva kivétel dobódik (**javax.ejb.NoSuchObjectException**). Az ábra egyetlen eddig nem érintett eleme a létező, de nem hivatkozott állapotban önmagába mutató állapotátmenet az ún. **home metódusokkal**. Ezekről elöljáróban annyit, hogy olyan üzleti metódusok, amelyek nem egy konkrét entity bean példányhoz kapcsolódnak, hanem többhöz (akár mindhez), ezért logikus módon a home interfészen keresztül hívhatók meg, vagyis nem szükséges az érintett példányokra referenciát szerezni.

Az entity beaneknek két altípusa van. **Bean Managed Persistence** (**BMP**) esetén a perzisztenciával kapcsolatos kódot, vagyis a megfelelő SQL-utasításokat a programozónak kell megírni a JDBC API segítségével, az életciklus megfelelő szakaszaihoz kapcsolódó metódusokban (**ejbCreate(), ejbLoad, ejbStore(), ejbRemove(), finder** metódusok). **Container Managed Persistence (CMP)** esetében ezzel szemben az EJB-konténer gyártójának eszközeivel generálódik minden JDBC-kód, így az nem a programozót terheli. A továbbiakban mindkét megoldást bemutatjuk. Egy egyszerű, könyveket tároló relációt (2.11. ábra) képezünk le BMP, majd CMP entity beanekre. Mindkét esetben mellőzzük a távoli interfészeket, mint azt később látni fogjuk, nem csak az egyszerűség kedvéért.



2.11. ábra *A könyveket tároló Book reláció*

2.1.3.1. Egy BMP entity bean

A lokális interfész: BMPBookLocal.java

A lokális interfészt a perzisztens attribútumok elérését és módosítását szolgáló getter/setter párosok alkotják.

```
package example;

public interface BMPBookLocal extends javax.ejb.EJBLocalObject {

  public Integer getId();
  public void setId(Integer id);

  public String getPublisher();
  public void setPublisher(String publisher);

  public String getTitle();
  public void setTitle(String title);
}
```

A lokális home interfész: BMPBookLocalHome.java

A lokális home interfész tartalmaz egy alkalmasan túlterhelt **create()** metódust, (tartalmazhatna többet is, pl. egy olyan verziót, ami nem minden attribútumhoz vár bemenő paramétert), illetve két finder metódust, amelyből szintén tetszőleges számú továbbit is lehetne definiálni. Az egyik elsődleges kulcs alapján, a másik cím alapján keres. Ennek megfelelően az első az egyetlen találat lokális interfészével, míg a másik a több lehetséges találatot reprezentáló entity bean lokális interfészét tartalmazó **Collection**-nel tér vissza. A keresés során felmerülő hibákat **FinderException** jelzi. Ennek alosztálya az **ObjectNotFoundException**, amely akkor dobódik, ha egyetlen találat helyett egyet sem talál a finder metódus, vagyis a példában a **findByPrimaryKey()** dobhat ilyet, míg a **findByTitle()** nem, mivel annak találat hiánya esetén üres **Collection**-nel kell visszatérnie.

```
package example;
import javax.ejb.FinderException;
import java.util.Collection;

public interface BMPBookLocalHome extends javax.ejb.EJBLocalHome {

  public example.BMPBookLocal create(
     Integer newId,
     String newTitle,
     String newPublisher)
     throws javax.ejb.CreateException;

  public example.BMPBookLocal findByPrimaryKey(Integer primaryKey)
     throws javax.ejb.FinderException;

  public java.util.Collection findByTitle(String title)
     throws FinderException;
}
```

Az implementációs osztály: BMPBookBean.java

Az implementációs osztály meglehetősen nagy méretű, ennek legfőbb okát a perzisztenciával kapcsolatos kódrészletek jelentik. Ezek az entity bean példánynak megfelelő relációbeli sor létrehozását (**ejbCreate()**), törlését (**ejbRemove()**), illetve a három memóriabeli példányváltozó és a nekik megfelelő adatbázisbeli értékek szinkronizálását (**ejbLoad(), ejbStore()**) és a keresést (**ejbFindByPrimaryKey(), ejbFindByTitle()**) végzik. Ezek mindegyike a szokásos módon használja a JDBC API-t:

- **Connection** objektum elkérése. Itt az egyetlen újdonság az, hogy az adatbázis-kapcsolat szerzése (a kódrészlet végén található **getConnection()** metódusban), követve a Java EE szemléletét, a JNDI segítségével történik.
- A **Connection** segítségével **PreparedStatement** létrehozása a megfelelő SQL-utasítást tartalmazó sztringgel, amely a paraméterek helyén kérdőjeleket tartalmaz.
- A paraméterek beállítása konkrét értékekre.
- A **PreparedStatement** végrehajtása, lekérdezés esetén az eredményeket tartalmazó **ResultSet** feldolgozása egy ciklusban.
- A **PreparedStatement** és a **Connection** bezárása, egy **finally** klózban, hogy a bezárás mind sikeres, mind sikertelen végrehajtás esetén megtörténjen.

A **Calculator** session beannél már találkozhattunk a **SessionContext** objektummal, ennek entity beanekben használt változata az **EntityContext**. Szerepe itt is az, hogy gatewayt biztosítson a konténer felé. Az **ejbLoad()** metódusban megfigyelhető ennek egyik fontos használati esete. Mikor a kon-

téner létrehoz egy példányt a bean osztályból, fel kell tölteni azt adatokkal, ezért meghívja az **ejbLoad()** metódust. Ahhoz, hogy tudjuk, melyik relációbeli sor adatait olvassuk be a memóriába, ismerni kell az entitás elsődleges kulcsát, amely az **EntityContext**-től kérdezhető le.

A metódusok elnevezésében itt is megfigyelhető a kötelező **ejb** prefix. Vagyis például a home interfész **create()** metódusát egy kliensből meghívva, a konténer az implementációs osztály **ejbCreate()** metódusát fogja meghívni. Fontos különbség vehető észre a home interfészben lévő **create()** metódus és az **ejbCreate()** visszatérési értéke között: az előbbi egy **BMPBookLocal**-lal tér vissza. Tudjuk a korábbiakból, hogy ezen keresztül a konténer ténylegesen egy általa generált, a **BMPBookLocal** interfészt megvalósító EJB-objektumot (vagy az EJB-objektummal kommunikálni tudó kliensoldali csonkot) ad vissza. Az implementációs osztálybeli **ejbCreate()**-ben ezt a generált osztályt nem tudjuk elérni, hogy visszaadhassuk, helyette az elsődleges kulccsal térünk vissza.

Hasonló az eset a finder metódusok esetén is. Például az **ejbFindByTitle()** visszatérési értéke egy olyan **Collection**, amely csak a találatok elsődleges kulcsait tartalmazza. Ha viszont a kliens a home interfészen keresztül a **findByTitle()**-t hívja, a visszakapott **Collection** a megtalált entitások lokális vagy távoli interfészeit tartalmazza. Ha ezeken a kliens végigiterál, minden egyes entitásra meghívódik az **ejbLoad()**, és így töltődnek be a talált elsődleges kulcsok alapján az adatok. Ez magában rejt egy jelentős teljesítményproblémát: ha egy keresés N találatot ad, akkor a finderben megírt **SELECT** megtalálja az elsődleges kulcsokat, utána pedig N db **ejbLoad()** hívás után lesznek az adatok elérhetők, vagyis összesen N+1 adatbázis-elérés szükséges. BMP entity beanek használata nélkül ez egyetlen lekérdezéssel is megoldható lenne. Ezen overhead és a megírandó JDBC-kód nagy mennyisége miatt a BMP helyett gyakorlatilag mindenhol a CMP javasolt.

```
package example;

import java.sql.*;
import java.util.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class BMPBookBean implements javax.ejb.EntityBean {

  //a perzisztens attribútumok
  private Integer id;
  private String title;
  private String publisher;

  //getter/setter párosok az attribútumok elérésére
  public Integer getId() {
     return id;
  }
```

```
    public void setId(Integer id) {
       this.id = id;
    }

    public String getTitle() {
       return title;
    }
    public void setTitle(String title) {
       this.title = title;
    }

    public String getPublisher() {
       return publisher;
    }
    public void setPublisher(String publisher) {
       this.publisher = publisher;
    }

    //az EJB-konténert az EntityContexten keresztül érjük el
    private javax.ejb.EntityContext myEntityCtx;

    //az EntityContext kezelése
    public javax.ejb.EntityContext getEntityContext() {
       return myEntityCtx;
    }

    public void setEntityContext(javax.ejb.EntityContext ctx) {
       myEntityCtx = ctx;
    }

    public void unsetEntityContext() {
       myEntityCtx = null;
    }

    //ejbActivate és ejbPassivate, mint session beaneknél
    //ebben a példában nem kell külső erőforrásokat elengedni, sem újra
    //megszerezni
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    /**
     * ejbLoad(): itt kell az osztályt feltölteni egy konkrét entitás
     * adataival
     * ez leggyakrabban relációs adatbázisból történik, mint itt is,
     * de más megoldás is lehetséges
     */
    public void ejbLoad() {

       //az ejbLoad()-ot a konténer azután hívja meg, miután az
       //objektumnak már meghívódott
       //a setEntityContext() metódusa, így le tudjuk kérdezni, hogy
```

```
        //mi az objektum
        //elsődleges kulcsa; ezt az adatot eltároljuk a tagváltozóban
        id = (Integer) myEntityCtx.getPrimaryKey();

        //a JDBC-híváshoz szükséges objektumok
        PreparedStatement pstmt = null;
        Connection conn = null;

        try {

              //adatbázis-kapcsolat nyitása
              conn = getConnection();

              //a megfelelő táblából kikeressük az adatokat az
              elsődleges kulcs alapján
              pstmt = conn.prepareStatement(
               "select title, publisher from books.book where id =?");

              pstmt.setInt(1, id.intValue());
              ResultSet rs = pstmt.executeQuery();
              rs.next();

              //az adatbázisbeli sor adataiból feltöltjük a
              //memóriabeli tagváltozókat
              title = rs.getString("title");
              publisher = rs.getString("publisher");

        } catch (Exception ex) {
              throw new EJBException(
                "Nem tölthető be a példány az adatbázisból "
                + id,
                ex);

        } finally {

              //JDBC-objektumok elengedése
              closeDBObjects(conn, pstmt);
        }
     }

     /**
      * ejbStore(): a memóriabeli adatok elmentését végzi a
        perzisztens tárolóba
      */
     public void ejbStore() {

        //a JDBC-híváshoz szükséges objektumok
        PreparedStatement pstmt = null;
        Connection conn = null;
        try {

              // JDBC-kapcsolat létrehozása
              conn = getConnection();
```

```
            //frissítjük a megfelelő adatbázisbeli sort a
            memóriabeli adatokkal
            pstmt = conn.prepareStatement(
                  "update books.book set title = ?, publisher = ?
                  where id = ?");

            pstmt.setString(1, title);
            pstmt.setString(2, publisher);
            pstmt.setInt(3, id.intValue());
            pstmt.executeUpdate();

      } catch (Exception ex) {
            throw new EJBException(
                  "A példány nem tárolható az
                  adatbázisban:" + id, ex);
      } finally {

            //JDBC-objektumok elengedése
            closeDBObjects(conn, pstmt);
      }
   }


   /**
    * ejbCreate(): új entitás létrehozása
    */
   public java.lang.Integer ejbCreate(Integer newId,
   String newTitle, String newPublisher)
      throws javax.ejb.CreateException {

      //a JDBC-híváshoz szükséges objektumok
      PreparedStatement pstmt = null;
      Connection conn = null;

      try {

            //az adatokat a memóriában is eltároljuk
            this.id = newId;
            this.title = newTitle;
            this.publisher = newPublisher;

            //JDBC-kapcsolat létrehozása
            conn = getConnection();

            pstmt = conn.prepareStatement(
                  "insert into books.book(id, title, publisher)
                  values (?, ?, ?)");

            pstmt.setInt(1, id.intValue());
            pstmt.setString(2, title);
            pstmt.setString(3, publisher);
            pstmt.executeUpdate();

            //az elsődleges kulccsal térünk vissza
```

```
            return id;
        } catch (Exception e) {
            throw new CreateException(e.toString());
        } finally {

            //JDBC-objektumok elengedése
            closeDBObjects(conn, pstmt);
        }
    }

    /**
     * ejbPostCreate(): minden ejbCreate()-hez tartozik egy
       ugyanolyan visszatérési értékű és paraméterlistájú
       ejbPostCreate(). Szerepéről később
     */
    public void ejbPostCreate(Integer newId, String newTitle,
    String newPublisher) throws javax.ejb.CreateException {
    }

    /**
     * ejbRemove(): az entitás eltávolítását végzi (a perzisztens
       tárolóból,
     * vagyis itt a relációs adatbázisból)
     */
    public void ejbRemove() throws javax.ejb.RemoveException {

        //szükségünk lesz az elsődleges kulcsra
        Integer key= (Integer) myEntityCtx.getPrimaryKey();

        //a JDBC-híváshoz szükséges objektumok
        PreparedStatement pstmt = null;
        Connection conn = null;
        try {
            // JDBC-kapcsolat létrehozása
            conn = getConnection();

            //a megfelelő sor törlése az adatbázisból
            pstmt = conn.prepareStatement(
                    "delete from books.book where id = ?");
            pstmt.setInt(1, key.intValue());

            //jelezzük, ha hiba volt
            if (pstmt.executeUpdate() == 0) {
                    throw new RemoveException("Nem távolítható el
                    az adatbázisból a példány" + key);
            }
        }
        catch (Exception ex) {
            throw new EJBException(ex.toString());
        }
        finally {

            //JDBC-objektumok elengedése
            closeDBObjects(conn, pstmt);
```

```
        }
    }

    /**
     * ejbFindByPrimaryKey(): elsődleges kulcs alapján keres az
       adatbázisban
     */
    public java.lang.Integer ejbFindByPrimaryKey(Integer key)
       throws javax.ejb.FinderException {

       //JDBC-objektumok
       PreparedStatement pstmt = null;
       Connection conn = null;

       try {

             //JDBC-kapcsolat létrehozása
             conn = getConnection();

             //Megkeressük az adatbázisban a megfelelő sort
             pstmt = conn.prepareStatement(
                    "select id from books.book where id = ?");

             pstmt.setString(1, key.toString());
             ResultSet rs = pstmt.executeQuery();
             rs.next();

             //az elsődleges kulccsal térünk vissza, ha nincs hiba
             return key;
             }
             catch (Exception e) {
                    throw new FinderException(e.toString());
             }
             finally {

                    //JDBC-objektumok elengedése
                    closeDBObjects(conn, pstmt);
             }

    }

    //egy másik finder metódus: az adott című könyveket keresi meg
    public Collection ejbFindByTitle(String title) throws
    FinderException {

       //JDBC-objektumok
       PreparedStatement pstmt = null;
       Connection conn = null;

       //ebbe fogjuk gyűjteni a megtalált entitások
       //elsődleges kulcsait
       Collection found = new ArrayList();

       try {
```