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.

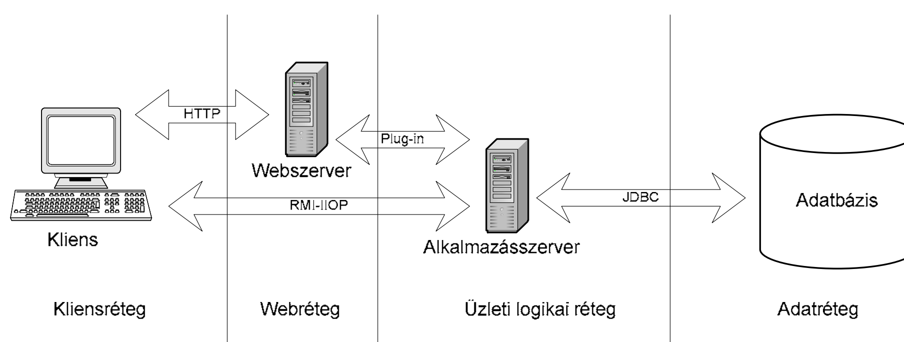
Technológia	Kapcsolódó middleware-szolgáltatás	Feje-zet
Java DataBase Connectivity (JDBC)	Perzisztencia	2
Java Persistence API (JPA)	Perzisztencia	2
Enterprise JavaBeans (EJB)	Többszálúság, perzisztencia, tranzakciókezelés, távoli elérés, aszinkron üzenetkezelés, biztonság	2, 6
JavaServlet	Távoli elérés (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 a CD-melléklet tartalmát ismerteti, és részletesen 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 a CD-melléklet 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.

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

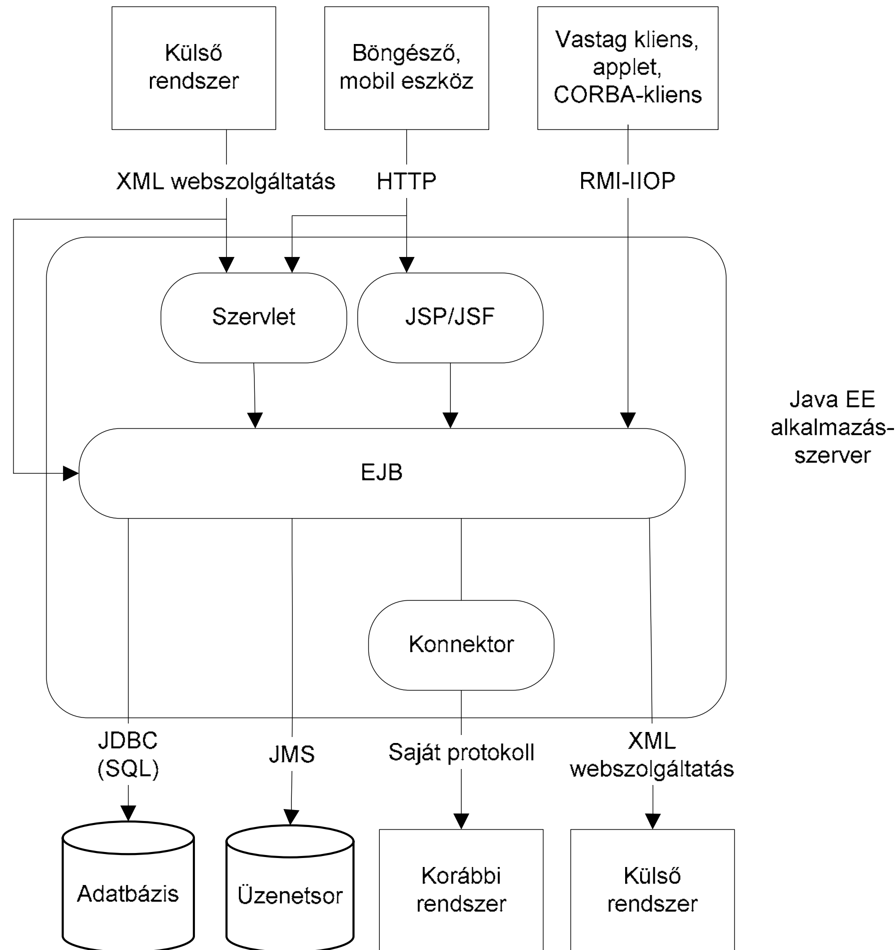
adatréteget felhasználva csökkenteni az egyik számla egyenlegét, és növelni a másikét. Fontos, hogy az „üzleti" szó itt nem pénzügyi értelemben jelent üzletet, egy hallgatói információs rendszerben például egy kurzus kiírása lehet egy üzleti funkció. Ez az a réteg, amelyet Java EE esetében egy alkalmazásszerverre 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 kommunikáció 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, alkalmazásszerverek 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 alkalmazás-

szervert 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.