

Debreceni Egyetem
Informatikai Kar

JAVA ENTERPRISE COMPUTING

Témavezető:

Dr. Fazekas Gábor

Egyetemi docens

Készítette:

Tündik Ferenc

Gazdaságinformatikus

Debrecen

2010

Tartalomjegyzék

Tartalomjegyzék	1
1. Bevezetés	2
2. A Java programozási nyelvről röviden	4
3. Java EE alapok.....	6
3.1. A Java EE alkalmazások felépítése	7
3.2. Komponensek és konténerek	9
3.3. A Java EE 5 API-k.....	10
3.4. Java EE 5 alkalmazások összeállítása, telepítése és csomagolása.....	13
4. Az Enterprise JavaBeans (EJB) Technológia és a Java Persistence API (JPA).....	15
4.1. EJB session bean komponensek	17
4.2. EJB üzenet-vezérelt bean-ek	24
4.3. A Java Persistence API és a JPA entitások.....	25
5. A Java Servlet Technológia.....	30
5.1. A Servletek és a hagyományos CGI programok összehasonlítása	30
5.2. A Servlet API.....	31
5.3. A HelloWorld Servlet.....	33
6. A JavaServer Pages (JSP) technológia	35
6.1. JSP jelölő elemek.....	36
6.2. Az egységesített EL	37
7. A JavaServer Faces (JSF) Technológia	39
Összefoglalás.....	51
Irodalomjegyzék	52
Függelék	53
A. Az adatbázist létrehozó SQL utasítások	53

1. Bevezetés

Tündik Ferenc vagyok, a Debreceni Egyetem Informatikai Karának gazdaságinformatikus hallgatója.

Szakedolgozatomban a Java platform vállalati programozást támogató kiadását a Java Enterprise Edition 5-öt mutatom be. Több oka is volt, hogy emellett a téma mellett döntöttem. A Java EE 5 platform olyan megoldásokat kínál a vállalati programozók számára, melyekkel korszerű, gyorsan és hatékonyan fejleszthető és karbantartható üzleti alkalmazások készíthetők. Bár a Java EE 5 nem az egyetlen vállalati programozási technológia, kétségtelenül az egyik leghatékonyabb és emellett az egyik legnépszerűbb is. Az Internet ezredfordulón tapasztalt gyors elterjedésével és dinamikus növekedésével együtt, egyre nagyobb igények mutatkoznak a Web-alapú alkalmazások iránt. A legfőbb ok, talán ezzel is kellett volna kezdenem, hogy engem is magával ragadt a Java programozás, és egyfajta hobbiként is tekintek rá.

A Java EE 5 platform sajátosságait egy általam készített adatbázis-alapú Webes könyvesbolt példaalkalmazásán keresztül fogom bemutatni. A példaalkalmazást a NetBeans IDE fejlesztőkörnyezet segítségével készítettem el, a Sun GlassFish Application Server v2 alkalmazás szerverre telepítettem, és adatbázis szerverként a Java DB adatbázis szervert használtam.

A dolgozat megírása során igyekeztem az egyes szakkifejezések magyar megfelelőjét használni, bár ez sokszor hiányos vagy erőltetett lehet, ezért az ilyen esetekben inkább az angol terminológiánál maradtam, és esetleg csak javaslatot tettem az általam megfelelőnek tartott magyar fordításra.

Szakedolgozatom első részében a Java programozási nyelv megszületésének körülményeiről írok néhány sort, majd a nyelv legfőbb sajátosságait foglalom össze röviden.

A következő részben a vállalati alkalmazások által támasztott legfőbb követelményeket és a legfontosabb sajátosságokat tárom fel, majd a Java EE alkalmazások tipikus felépítését, architektúráját és telepítését mutatom be. Röviden összefoglalom a Java EE platform által nyújtott alkalmazásprogramozási interfészeket (API-kat) is, melyeket a későbbi fejezetekben mutatok be részletesen.

Ezt követően a platform üzleti logikájának megvalósítását támogató technológiákat, az EJB és a JPA technológiákat mutatom be először elméleti megközelítésben, majd az alkalmazásból vett gyakorlati példákon keresztül is.

A következő fejezetekben pedig a prezentációs réteg technológiáit ismertetem. Először a Java szerver-oldali programozásának alapját képező Servlet technológiát, majd az erre épülő JSP technológiát, végül a legújabb és egy más megközelítést alkalmazó támogató keretrendszert, a JSF technológiát fogom bemutatni.

2. A Java programozási nyelvről röviden

Az 1990-es évek elején James Gosling vezetésével a Sun elindított egy projektet, melynek célja az volt, hogy a cég teret hódítson a felhasználói elektronikai piac egy új, az úgynevezett okos, processzorral vezérelt, programozható készülékeket (pl. set-top-box) alkalmazó területén. Ezen készülékek programozásához olyan architektúra-független technológiára volt szükség, mely lehetővé tette kész programok hálózaton keresztül történő letöltését a készülékbe és ezeknek a programok a megbízható futtatását. Kezdetben a C++ nyelvvel próbálkoztak, de ez a nyelv csak úgy, mint a többi, nem volt megfelelő a projekt célkitűzései által támasztott követelmények megvalósításához, így elhatározták, hogy egy új programozási nyelvet készítenek. Sajnos a felhasználói elektronikai alkalmazások lassabban fejlődtek, mint ahogy azt várták, így a projekt lassan zsákutcába futott volna, de ekkor kezdődött el az Internet rohamos terjedése. Szerencsére észrevették, hogy az Internetes hálózat hasonló körülményeket teremt és hasonló elvárásokat támaszt az új programozási technológiával szemben, így a projekt megmenekült. Az új nyelv első nyilvános implementációja 1995-ben jelent meg, neve több változtatás után Java 1.0 lett.

A Java nyelv megalkotása során megfogalmazott legfőbb célkitűzések az alábbiak voltak:

Legyen egyszerű, objektum-orientált: A nyelv szintaktikája és szemantikája a sokak által ismert C, illetve C++ nyelvekhez hasonlít, de azokénál lényegesen egyszerűbb. Egy modern programozási nyelv manapság szinte el sem képzelhető anélkül, hogy ne támogatná az objektum-orientált programozási szemléletet. A Java is ennek tükrében objektum-orientált nyelv, lényegében a C++ objektum-orientált tulajdonságait tartalmazza néhány eltéréstől eltekintve.

Legyen architektúra-független és hordozható: Napjaink hálózatait változatos hardver- és szoftver architektúrájú számítógépek és eszközök alkotják. A szoftverfejlesztést nagymértékben megkönnyítené, ha a forráskódból előállított program bármely architektúrán azonosan futna, ezért a Java nyelv nem tartalmaz architektúra- vagy implementációfüggő elemeket. Ezt azzal érték el, hogy a forráskódot a fordítóprogram nem konkrét gépi kódra, hanem egy közbülső kódra, az úgynevezett byte kódra fordítja le, melyet egy képzeletbeli hardver, a virtuális gép értelmezni és futtatni tud. Így egy új architektúrára csak a virtuális

gépet megvalósító programot kell megírni és bármely Java program változtatás nélkül futtatható rajta.

Legyen interpretált, dinamikus és többszálás: A Java nyelven megírt programokat sokan lassúnak tartják, mivel az interpretált végrehajtás lassabb, mint a gépi kódé, de a bevezetett újításoknak köszönhetően a programok végrehajtási ideje jelentősen csökkent. Ilyen újítás többek között az ún. Just-in-time compiler (fordító), jobb kódanalízis és a HotSpot, amely magában a Java Virtual Machine-ben hajtott végre optimalizálást. Az interpretált jellegnek köszönhetően azonban a programfejlesztési ciklus nagymértékben felgyorsult, mivel elég csak a megváltozott állományokat lefordítani, nincs szükség a forrásállományok közötti függőségek feltérképezésére. A Java nyelvi szinten biztosítja a többszálás programozás lehetőségét. Ennek révén a program több párhuzamosan végrehajtható részre, vezérlési szálra bontható, melyek egymással kommunikálhatnak, így a processzor jobban kihasználható, az egyes feladatok áttekinthetőbben, könnyebben megvalósíthatók.

Legyen robusztus és biztonságos: Egy programozási nyelv robusztus, ha megakadályozza vagy kiszűri a programozási hibákat. Ez nyelvi szinten szigorú, statikus típusosságban jelenik meg, minden adatnak fordításkor jól definiált típusa van. Mivel a Java nyelvben nincsenek mutatók, rengeteg programozási hibától megszabadulhatunk (pl. null pointer hivatkozás). A dinamikus szemétygyűjtés automatikusan felszabadítja az elérhetetlen, nem használt memóriát, habár a logikai memóriaszivárgástól (memory leak) nem véd meg, amikor a memóriaterületre mutat referencia, de az nincs használva. A biztonság a robusztussággal kezdődik, a fordító csak megfelelően viselkedő programot készít el. A betöltött programokat a betöltő konzisztencia-ellenőrzője elemzi, majd a virtuális gép felügyelete alá kerülnek, mely ellenőrzi, hogy a program csak engedélyezett tevékenységet hajt-e végre.

A fent leírt előnyös tulajdonságainak köszönhetően a Java nyelv az Internet-alapú alkalmazások és hálózaton keresztül kommunikáló eszközökre írt szoftverek egyik legnépszerűbb nyelvává vált. 1998-ban a Java újabb verziójának (J2SE 1.2) megjelenésével együtt két új verzió is megjelent, melyek a standard kiadásra épülnek. A J2ME kiadás lehetővé teszi alkalmazások készítését olyan kicsi, korlátozott memóriájú és teljesítményű eszközökre, mint a mobiltelefonok vagy a PDA-k. A J2EE kiadással ellentétes irányba

mozdultak el, nagyteljesítményű, elosztott, hálózati és Web-alapú szerver-oldali alkalmazások hatékony fejlesztését téve lehetővé.

Habár a legújabb vállalati Java kiadás a Java EE 6, ezzel kapcsolatban még nagyon kevés szakirodalom áll rendelkezésre, ezért szakdolgozatom megírása során a Java EE 5 kiadást fogom ismertetni és használni.

3. Java EE alapok

A vállalati alkalmazásokat üzleti problémák megoldására készítik. Ez a folyamat általában magában foglalja az üzleti adatok biztonságos tárolását, elérését és módosítását. Ilyen üzleti adat lehet például vásárló lakhelye, hitelkártya száma, repülőjegy foglalás adatai, stb. Ezek a vállalati alkalmazások többféle felhasználói interfésszel rendelkeznek, például egy Webes felület a vásárlók részére és egy grafikus felhasználói interfész (GUI) a cég irodájában dolgozó alkalmazottak részére. Egyúttal távoli rendszerekkel is tudniuk kell kommunikálni, különböző adattárakban tárolt adatokat kell konzisztensen kezelniük, és biztosítaniuk kell, hogy a rendszer az üzlet által meghatározott szabályok szerint működjön. Fontos az is, ha az üzletmenetben változás következik be, akkor a rendszert gyorsan és hatékonyan lehessen a változásokhoz igazodva módosítani.

Amikor egy GUI alkalmazást készítünk, nem kell az alapoktól kezdeni, például hogyan lehet egy pixelt a képernyőre kirajzolni, építhetünk olyan GUI könyvtárakra, mint a Swing. Ugyanígy nem kell teljesen az alapoktól kezdenünk egy vállalati alkalmazást sem. A vállalati programozók alkalmazásaikat az úgynevezett alkalmazás szerverekre építik. Ilyen alkalmazás szerverek például a GlassFish Application Server v2 (Sun), JBoss (RedHat), Geronimo (Apache Software Foundation), Weblogic (BEA), Websphere (IBM). Ezek az alkalmazás szerverek olyan szolgáltatásokat nyújtanak, melyek minden vállalati alkalmazás esetében alapvető fontosságúak, mint például hálózati kommunikáció, adatbázishoz való kapcsolódás, tranzakció-menedzsment, biztonsági szolgáltatások, stb. Így a programozó a konkrét üzleti logika megvalósítására koncentrálhat, nem kell alapvető kérdésekkel foglalkoznia, azokat az alkalmazás szerver elérhetővé teszi számára.

Tehát a Java Platform, Enterprise Edition 5 olyan szabványt definiál, mely lehetővé teszi komponens-alapú, többretegű vállalati alkalmazások fejlesztését. Ahogy a neve is utal rá, a Java EE a nagyméretű üzleti rendszerekre összpontosít. Azok a szoftverek, melyek ilyen rendszerekre készülnek, nem egyetlen PC-n futnak, jelentősen nagyobb számítási

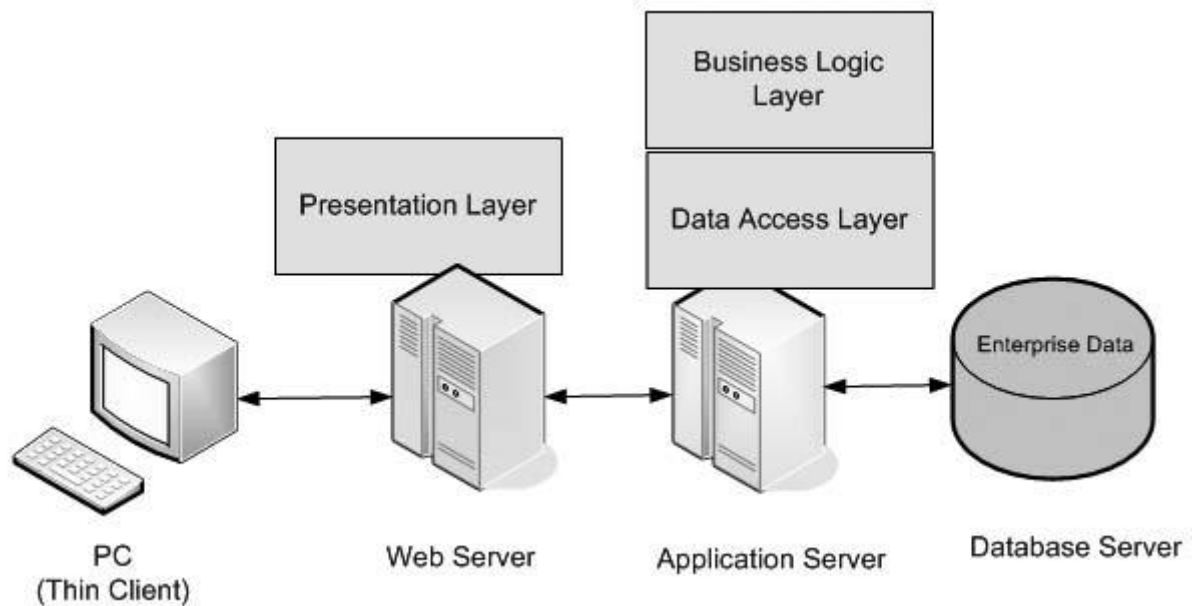
teljesítményt és adatátviteli sebességet igényelnek. Ezen okok miatt a szoftvert funkcionális részekre kell osztani és a különböző részeket a megfelelő hardverre telepíteni. Ez az alapja az elosztott adatfeldolgozásnak. A Java EE standardizált komponensek egy olyan kollekcióját nyújtja, melyek lehetővé teszik a szoftverek telepítését, olyan standard interfészeket tartalmaznak, melyek definiálják, hogy a különböző szoftver modulok hogyan kapcsolódnak egymáshoz és olyan standardizált szolgáltatásokat nyújtanak, melyek azt definiálják, hogy ezek a modulok hogyan kommunikálnak egymással.

3.1. A Java EE alkalmazások felépítése

A Java EE platform a vállalati alkalmazásokhoz az elosztott, többretegű alkalmazásmodellt használja [3] [8] [11]. Ennek lényege, hogy az alkalmazást több kisebb funkcionális részre osztjuk, melyek egyidejűleg futnak különböző számítógépeken és hálózaton keresztül kommunikálnak egymással. Ezeket a kisebb részeket rétegeknek nevezzük. Minden egyes rétegnek megvan a saját jól meghatározott feladata. A legtöbb alkalmazás három rétegre osztható:

- A prezentációs réteg, mely a felhasználói felületért és az adatok érvényesítéséért felelős. Lehetővé teszi, hogy a felhasználók kapcsolatba léphessenek az alkalmazással. Futhat a kliens gépen vagy a Java EE szerveren.
- Az üzleti réteg, mely az üzleti logika megfelelő végrehajtásáért felel. Kapcsolatban áll a prezentációs és az adathozzáférési réteggel is. A felhasználók kéréseit dolgozza fel, melyek a prezentációs rétegből érkeznek. A Java EE szerveren fut.
- Az adathozzáférési réteg felelős a vállalati információs rendszerben (EIS) tárolt adatok eléréséért és manipulálásáért. Felosztható az adatbázis rétegre, amely általában egy objektumok tárolására használt relációs adatbázis, és a perzisztencia rétegre, mely az üzleti réteget kapcsolja össze a mögöttes adatbázissal. Egyes megközelítésekben a perzisztencia réteget az üzleti réteg részének tekintik. Futhat a Java EE szerveren vagy az EIS (adatbázis) szerveren.

Az egyes rétegek további rétegekre bonthatók, de lehetőleg ne bonyolítsuk túl az alkalmazás szerkezetét, mivel a nem megfelelő tervezés könnyen a teljesítmény rovására mehet.



1. ábra Egy tipikus Java EE alkalmazás felépítése

Az 1. ábrán látható egy rétegelt, elosztott Web-alapú Java EE alkalmazás felépítése. Ebben a modellben, ami lényegében a kliens-szerver architektúra továbbfejlesztett változata, a PC-nek csak egy egyszerű böngészőre van szüksége a szervertől érkező prezentációs tartalom megjelenítéséhez. A Web-szerver felelős a prezentációs tartalom generálásáért. Az alkalmazás szerveren található az üzleti logika és az adathozzáférési réteg. Az üzleti logika távoli metódushívásokkal érhető el, így támogatja az önálló kliens (Java konzol) alkalmazásokat is. Az alkalmazás adatai a vállalati információs rendszerből érhetők el, ami általában egy relációs adatbázis.

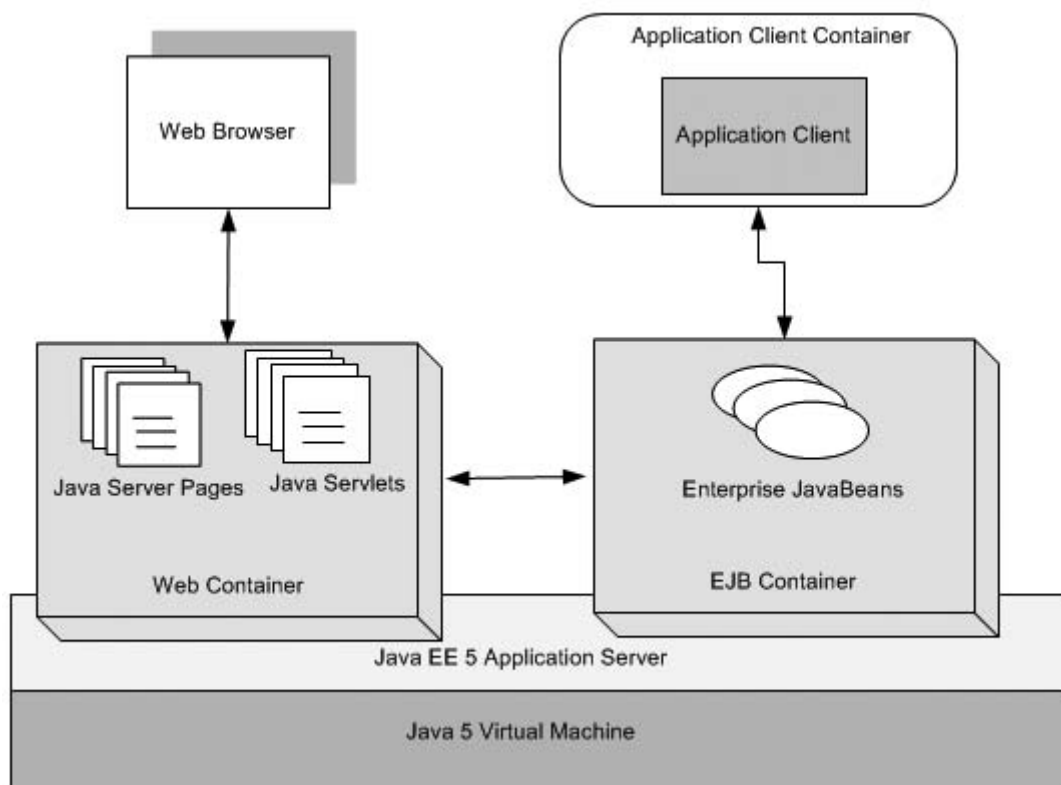
A többrétegű alkalmazások számos kedvező tulajdonsággal rendelkeznek. Az alkalmazás karbantartási költségei alacsonyan tarthatók. Ha az üzleti logika változik, a módosítást csak egy helyen, az alkalmazás szerveren kell végrehajtani, nem kell minden kliens programot frissíteni, ami egyúttal növeli az alkalmazás konzisztenciáját is. Az üzleti adatok és szabályok megoszthatók az egyes alkalmazások között, ezzel hatékonyan és olcsón megoldható a különböző alkalmazások közötti együttműködés. A rétegelt architektúrából következik, hogy az egyes rétegek csak bizonyos feladatokat látnak el, így az alkalmazás gondos tervezéssel nagyobb teljesítményen üzemelhet. Azzal, hogy az alkalmazás üzleti logikája egy helyen található, jelentősen növelhető az alkalmazás biztonsága is.

3.2 Komponensek és konténerek

A Java EE alkalmazások komponensekből épülnek fel [3] [8] [11]. A Java EE komponensek olyan funkcionális programegységek, melyek a Java EE alkalmazásokba a kapcsolódó osztályaikkal és fájljaikkal együtt épülnek be és kommunikálnak más komponensekkel. A Java EE komponenseket a Java programozási nyelven írjuk, és ugyanúgy fordítjuk őket, mint a többi programot. A standard Java osztályoktól az különbözteti meg őket, hogy a Java EE specifikáció szerinti felépítésűeknek kell lenniük, és a Java EE szerver futtatja és kezeli őket. A Java EE specifikáció által definiált komponensek:

- Alkalmazás kliensek és appletek, a kliens gépen futnak.
- Java Servlet, JavaServer Pages™ (JSP™), JavaServer Faces (JSF™) technológiájú komponensek olyan Web komponensek, melyek a Java EE szerveren futnak.
- Enterprise JavaBeans™ (EJB™) komponensek, olyan üzleti komponensek, melyek a Java EE szerveren futnak.

Ezeket a komponenseket a későbbiekben fogom részletesen bemutatni.



2. ábra A Java EE platform architektúrája

A Java EE platform konténereken keresztül biztosítja az alapvető rendszerszolgáltatásokat. A komponensek futtató környezetül a konténerek szolgálnak, a komponensek a konténereken belül helyezkednek el és jól definiált interfészeket keresztül kommunikálnak a Java EE infrastruktúrával. Olyan alacsony szintű szolgáltatásokat nyújtanak, mint például biztonsági szolgáltatások, tranzakció kezelés, életciklus kezelés, objektumkeresés és gyorstárazás, perzisztencia vagy hálózati szolgáltatások. Ha egy Web-komponenst vagy egy EJB komponenst használni szeretnénk, egy Java EE modulba és a megfelelő konténerbe kell helyoznunk. Ez a folyamat magában foglalja olyan konténer beállítások végrehajtását, melyek az egyes komponensekre érvényesek a Java EE alkalmazáson belül. A Java EE specifikáció az alábbi konténereket határozza meg:

- A Java EE szerver: A Java EE alkalmazások futtató környezete. Tartalmazza az EJB és a Web konténert.
- Enterprise JavaBeans (EJB) konténer: Egy Java EE alkalmazás EJB komponenseit kezeli. Az EJB-k és az EJB konténer a Java EE szerveren futnak.
- Web konténer: Egy Java EE alkalmazás Servlet, JSP és JSF komponenseit kezeli. A Web komponensek és a Web konténer a Java EE szerveren futnak.
- Alkalmazáskliens konténer (ACC): Az alkalmazásklienseket kezeli, a kliens gépen fut.
- Applet konténer: Az appletek végrehajtását kezeli. Egy böngészőből és egy Java Plug-in-ből áll, melyek a kliensen futnak.

3.3 A Java EE 5 API-k

A Java EE 5 platform az alábbi API-kat (alkalmazásprogramozási interfészeket) nyújtja a programozók számára:

- Java Servlet Technology: Lehetővé teszi, hogy egyszerű és konzisztens módon kiterjesszük és fokozzuk az alkalmazást futtató szerver nyújtotta lehetőségeket. A Servlet dinamikus tartalmat generál és kérés/válasz elven működik. A Servlet bármilyen típusú kérésre tud válaszolni, azonban a gyakorlatban a HTTP-alapú kérések kezelésére szolgáló Servletek a gyakoriak, melyek a Web-szervereken futnak.
- JavaServer Pages Technology (JSP): Olyan Web technológia, mely lehetővé teszi Java kód beágyazását olyan szöveg alapú dokumentumokba, mint a HTML vagy XML dokumentumok. A JSP oldalak sablon adatot, egyedi elemeket, script nyelveket és szerver-

oldali Java objektumokat használnak a dinamikus tartalom generálására. Az alkalmazás szerverek a JSP oldalakat Servletekké fordítják le.

- **JavaServer Pages Standard Tag Library (JSTL):** Olyan standardizált egyedi JSP jelölő elemek (tag-ek) könyvtára, melyek a legtöbb alkalmazásban gyakran használt alapfunkciók (pl. feltételes elágazások, iterátorok, adatbázis-elérés SQL-el, gyakran használt függvények) egyszerű használatát biztosítja.

- **JavaServer Faces Technology (JSF):** A JSF olyan komponens alapú technológia, mely lehetővé teszi server-oldali felhasználói interfészek készítését. A JSF a JSP és Servlet technológiára épül.

- **Enterprise JavaBeans Technology (EJB):** A Java EE platformban az üzleti logika megvalósítására szolgál. Az EJB architektúra lehetővé teszi skálázható, biztonságos, elosztott üzleti alkalmazások készítését. Két fajta EJB létezik: a session bean, mely a klienssel való interakciót testesíti meg, és a message-driven (üzenet-vezérelt) bean, mely aszinkron üzenetek fogadására szolgál. A korábbi EJB specifikációban létező entity (entitás) bean-t felváltotta a JPA entitás.

- **Java Transaction API (JTA):** A Java EE platform standard tranzakció-kezelő API-ja.

- **Java Persistence API (JPA):** A Java EE platform standard API-ja az adattároláshoz és adathozzáféréshez. Az objektum-orientált modellt az ún. objektum-relációs leképezéssel (ORM) kapcsolja össze a relációs adatbázismodellel.

- **Java Naming and Directory Interface (JNDI):** Ez az API, amely a Java SE platformhoz tartozik, lehetővé teszi, hogy névvel ellátott objektumokat és erőforrásokat érjünk el olyan könyvtár rendszerekből, mint az LDAP vagy a DNS. Ezek az objektumok lehetnek a rendszer által nyújtott vagy a felhasználó által definiált objektumok. A Java EE 5 platformban a sokszor igen körülményes JNDI hívások az injektálásnak köszönhetően szinte láthatatlan módon mennek végbe.

- **Java Message Service API (JMS):** A JMS API standard módot biztosít aszinkron üzenetek küldésére és fogadására a Java EE komponensek számára, és lehetőséget biztosít egyedi üzenetkezelő API készítéséhez is. A point-to-point (P2P) és a publish-subscribe üzenetkezelési modellt támogatja.

- **JavaMail API:** Ez az API lehetővé teszi email-ek küldését.

- **JavaBeans Activation Framework (JAF):** A Java SE platformban található JAF egy olyan keretrendszert nyújt, mely lehetővé teszi különböző MIME típusú adatok feldolgozását. A JavaMail API használja.

- **Java API for XML Processing (JAXP):** Ez az API is a Java SE platform része, mely XML dokumentumok feldolgozásához nyújt segítséget. Támogatja a DOM API-t, a SAX API-t és az XSLT-t.

- **Java API for XML Web Services (JAX-WS):** A JAX-WS olyan Webes szolgáltatásokhoz nyújt támogatást, melyek a JAXB API-t használják XML adatok Java objektumokhoz kapcsolásához.

- **Java Architecture for XML Binding (JAXB):** XML adatok Java objektumokhoz kapcsolását teszi lehetővé.

- **SOAP with Attachments API for Java: (SAAJ):** Erre az alacsony szintű API-ra épül a JAX-WS és a JAXR specifikáció.

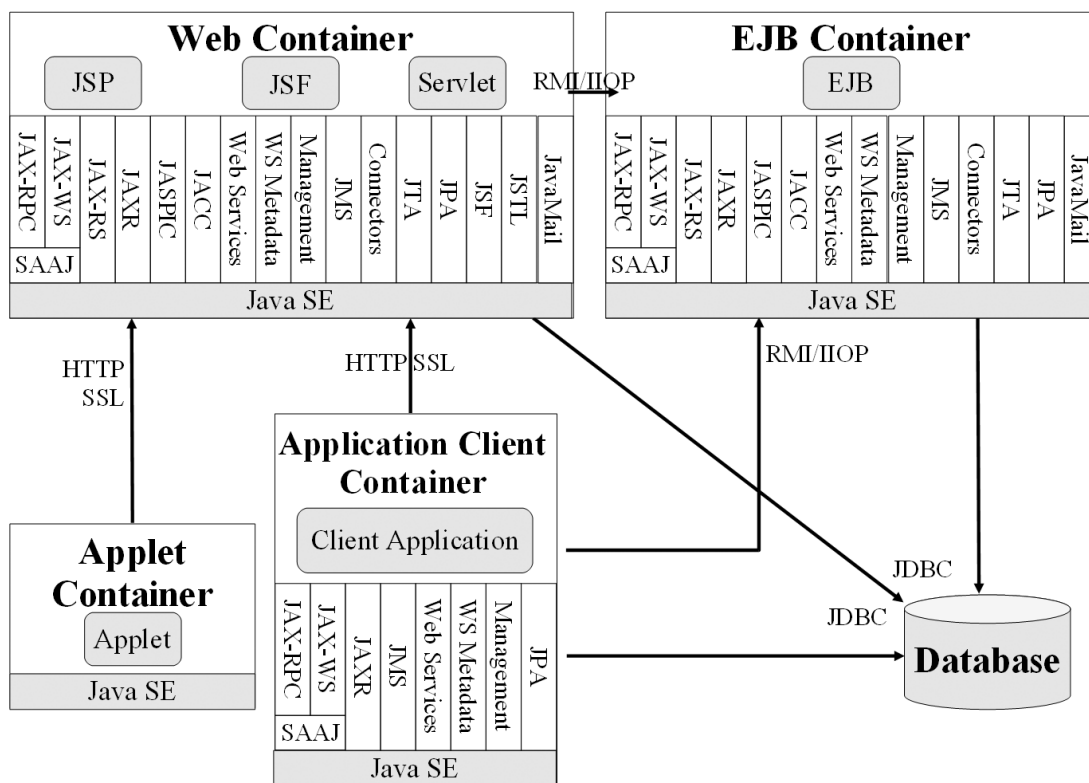
- **Java API for XML Registries (JAXR):** Lehetőséget biztosít üzleti és általános célú Webes jegyzékek eléréséhez.

- **J2EE Connector Architecture (JCA):** Hozzáférést tesz lehetővé vállalati információs rendszerekhez (EIS).

- **Java Database Connectivity API (JDBC):** A JDBC API lehetővé teszi Java EE alkalmazások hozzákapcsolódását szinte bármilyen táblázatos adatforráshoz és SQL parancsok hívását a Java metódusokban.

- **Java Authentication and Authorization Service (JAAS):** A JAAS API biztonsági szolgáltatásokat nyújt a Java EE alkalmazások számára. Ilyen biztonsági szolgáltatás a felhasználók hitelesítése és azonosítása.

A 3. ábrán látható, hogy az egyes Java EE konténerek mely API-kat támogatják és milyen protokollokon keresztül kommunikálnak egymással.

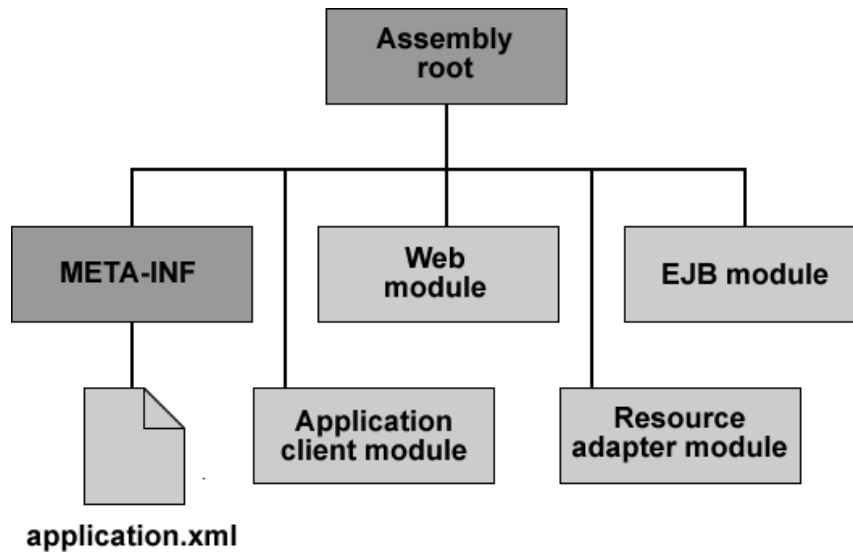


3. ábra A Java EE 5 konténerek és az általuk támogatott API-k

3.4 Java EE 5 alkalmazások összeállítása, telepítése és csomagolása

A Java EE alkalmazásokat egy vagy több standard egységbe, ún. modulokba csomagoljuk mielőtt egy Java EE szerverre telepítjük [11]. Minden ilyen modul tartalmaz egy vagy több komponenst és egy opcionális telepítés leíró (deployment descriptor) fájlt, mely a modul tartalmát írja le. A telepítés magában foglalja olyan környezet-specifikus információk megadását, mint az alkalmazás jogosult felhasználói vagy a helyi adatbázis elérési útja.

A Java EE 5 alkalmazásokat Enterprise Archive (EAR) fájlokba csomagoljuk, melyek standard Java Archive (JAR) fájlok .ear kiterjesztéssel (4. ábra). Az EAR fájl Java EE modulokat és telepítés leíró fájlokat tartalmaz. A telepítés leíró fájl egy XML dokumentum .xml kiterjesztéssel, mely egy alkalmazás, modul vagy komponens telepítési beállításait tartalmazza. A telepítés leíró információk deklaratív jellegűek, így azok a forráskód módosítása nélkül változtathatók.



4. ábra Az EAR fájl szerkezete

Két fajta telepítés leíró fájl van az egyik a Java EE telepítés leíró, amelyet a Java EE specifikációban határoztak meg és olyan telepítési beállítások konfigurálása használható, melyek bármely Java EE szerverre érvényesek. A másik a futás alatti telepítés leíró, mely implementáció specifikus paraméterek beállítására használható.

Egy Java EE modul az egy konténertípushoz tartozó komponenseket és az adott típushoz tartozó telepítés leíró fájlt tartalmazza. Az alkalmazás telepítés leíró nélküli Java EE modul telepíthető önálló modulként is. Négy fajta Java EE modul létezik:

- Web modul: Servlet osztály fájlokat, JSP fájlokat, segédosztály fájlokat, HTML, CSS és képfájlokat (pl. GIF) és egy Web alkalmazás telepítés leíró fájlt tartalmaz. A Web modult olyan JAR fájlba csomagoljuk, melynek a kiterjesztése .war (Web Archive).
- EJB modul: EJB-k osztály fájljait és egy EJB telepítés leíró fájlt tartalmaz. Az EJB modult JAR fájlba csomagoljuk .jar kiterjesztéssel.
- Alkalmazáskliens modul: Osztály fájlokat és egy alkalmazáskliens telepítés leíró fájlt tartalmaz. Az alkalmazás kliens modult JAR fájlba csomagoljuk .jar kiterjesztéssel.
- Erőforrás adapter modul: Tartalmazza az összes Java interfészt, osztályt, natív könyvtárakat és egyéb dokumentációkat és az erőforrás adapter telepítés leíró fájlt. Együtt implementálják a Connector architektúrát egy konkrét vállalati információs rendszerhez (EIS). Az erőforrás adapter modult JAR fájlba csomagoljuk .rar (Resource Adapter Archive) kiterjesztéssel.

4. Az Enterprise JavaBeans (EJB) Technológia és a Java Persistence API (JPA)

Az Enterprise JavaBeans Technológia egy olyan platform, mely lehetővé teszi hordozható, újrafelhasználható és skálázható üzleti alkalmazások készítését Java programozási nyelven [2] [6] [7]. Az üzleti logika hatékony megvalósítását az EJB Technológián keresztül támogatja a Java EE platform. A fő hajtóerő az EJB mögött az volt, hogy egy olyan komponens modellt vagy keretrendszert hozzanak létre, melyben nem kell mindig újraprogramozni olyan alapvető üzleti szolgáltatásokat, mint a tranzakció kezelés, a biztonság vagy az automatikus perzisztencia, hanem ezeket a szolgáltatásokat a rendszer alapvető szinten biztosítsa az alkalmazás készítői számára.

Az EJB Technológiát célszerű ha az alkalmazás az alábbi követelmények valamelyikét megköveteli:

- Az alkalmazásnak skálázhatónak kell lennie: Hogy kezelni tudjuk a felhasználók növekvő számát az alkalmazás komponenseit több szerverre kell elosztani. Azonkívül hogy az EJB komponensek különböző gépeken képesek futni, ez a kliensek számára láthatatlan módon történik.
- Az alkalmazásnak biztosítania kell az adatok integritását: Az EJB komponensek támogatják a tranzakció kezelést, azt a mechanizmust, mellyel a megosztott objektumok egyidejű hozzáférése kezelhető.
- Az alkalmazás kliensei többfélék: Néhány sornyi kóddal a távoli kliensek egyszerűen megtalálhatják az EJB komponenseket. Ezek a kliensek változatos típusúak és formájúak lehetnek.

Az EJB, mint keretrendszer biztosítja az EJB bean-ek működéséhez a támogató környezetet. Ez a támogató környezet az EJB konténer, mely olyan szolgáltatásokat nyújt, mint a tranzakció kezelés és a biztonsági szolgáltatások, erőforrás tárolók alkalmazása és az erőforrások gyors tárolása, a komponensek életciklusához tartozó szolgáltatások vagy a konkurencia szolgáltatások.

Az EJB, mint egy komponens modell két fajta komponenst biztosít az üzleti logika megvalósítására:

- Session bean (Munkafolyamat bean): Az üzleti szolgáltatások végrehajtásáért felelős, illetve a tranzakció kezelést és az adatokhoz való hozzáférést vezérli.
- Message-driven bean (Üzenet-vezérelt bean-MDB): A külső eseményekre aszinkron módon válaszoló objektumok.

Az EJB 3.0 lényeges változtatásokat vezetett be a korábbi EJB specifikációkhoz képest. A korábbi verziókkal ellentétben az EJB specifikációból eltűntek a perzisztencia kezeléséért felelős entitás bean-ek, és az API jelentősen egyszerűbben használhatóvá vált. A perzisztencia kezelésére egy teljesen új API-t, a Java Persistence API-t, hoztak létre.

Az EJB és a JPA specifikációk alapvető sajátosságai:

- Deklaratív meta-adatok: Lehetővé teszik, hogy a fejlesztők a deklaratív módon határozzák meg a komponensek viselkedését, szemben a programozás szintű specifikálással. Erre a JDK 5.0-ban bevezetett annotációk és az XML leírók tökéletesen alkalmasak.
- Kivételeken alapuló konfigurálás: Intelligens alapértelmezett adatok révén a komponensnek csak azokat a tulajdonságait kell explicit módon megadni, melyek az alapértelmezésektől eltérnek.
- Dependency Injection (DI): A Spring keretrendszerből átvett függőségi beinjektálás segítségével inicializálhatunk adatokat az adott objektum példányra vonatkozóan, még mielőtt az objektum példányt a kliens rendelkezésére bocsátanánk. A DI segítségével megkímélhetjük a komponenst a költséges erőforrás lekérések (JNDI hívások) végrehajtásától, ha azokra a bean használata során valószínűleg nem lesz szükség..
- Skálázhatóság: A nagyméretű alkalmazások megkövetelik, hogy a terheltség növekedésével együtt az alkalmazás rugalmasan skálázható legyen. Ezen követelmények kielégítésére használható az erőforrás tárolók (resource pool) alkalmazása és a perzisztencia gyorsítótár (persistence cache).
- Tranzakció kezelés: A Java Transaction API (JTA) egy standardizált alkalmazásprogramozási interfészt biztosít az elosztott tranzakciók kezeléséhez, és az EJB konténer úgy működik, mint az EJB komponensek JTA tranzakció-kezelője. Ennél fogva az EJB specifikáció lehetővé teszi, hogy deklaratív módon határozzuk meg a konténer által kezelt komponensek tranzakciós tulajdonságait.

- Több-felhasználós biztonság: Metódus-szintű hozzáférés-védelem adható be deklaratív módon, mellyel felhasználó- vagy szerep-szintű privilégiumok adhatók meg.
- Hordozhatóság: A specifikációnak megfelelő komponensek szabadon telepíthetők bármely az EJB specifikációt implementáló alkalmazás szerverre.
- Újrafelhasználhatóság: Az EJB komponensek lazán kapcsolt komponensek, így eltérő alkalmazásokban is felhasználhatók.
- Perzisztencia: Az EJB entitás bean-eket felváltották a JPA entitások, melyek egyszerű régi Java objektum (POJO plain old Java object) osztályok. Egy entitás osztály megfeleltethető az adatbázis egy táblájának, míg az egyes entitás példányok a táblázat sorainak.

4.1. EJB session bean komponensek

Az EJB session bean-ek olyan Java komponensek, melyekkel a konkrét üzleti folyamatokat és a felhasználói interakciókat modellezhetjük [2] [6] [7]. A session bean a kliens számára végez munkát, a komplex, számításigényes feladatokat a szerveren hajtva végre. A session bean-ek átmeneti jellegűek, viszonylag rövid életűek, mivel nem tároljuk őket adatbázisokban vagy más állandó jellegű fájlrendszerekben. A kliens a bean-nel egy vagy több a bean-ben definiált metódus hívásán keresztül lép kapcsolatba. Az egyes bean-ek nincsenek megosztva a kliensek között. A kliens lehet például Web-kliens, mint egy Servlet vagy egy JSP oldal, vagy egy önálló Java alkalmazás.

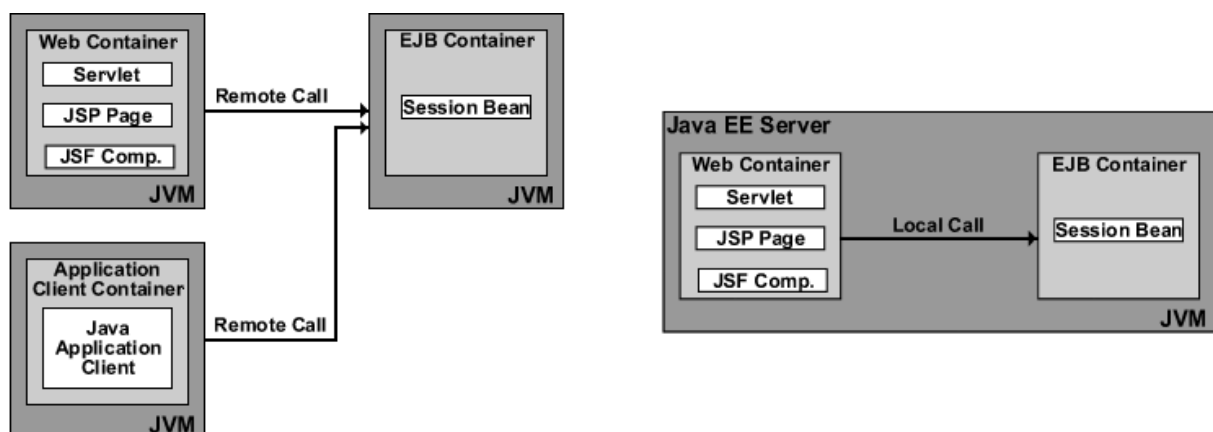
A session bean-eknek két fajtája van aszerint, hogy tárolnak-e interakciós adatokat az egyes metódushívások között:

- Stateful session bean (Állapottal rendelkező session bean): A bean állapota a kliens metódushívásai között automatikusan megmarad. Minden egyes klienshez külön bean példány van hozzárendelve. Tipikusan ilyen állapot-érzékeny üzleti eset a bevásárlókosár egy Webes áruházhoz, melyet én is stateful session bean-nel valósítottam meg.
- Stateless session bean (Állapotmentes session bean): Ahogy a neve is utal rá, az állapotmentes session bean nem tárol interakciós állapotot és olyan szolgáltatások modellezésére használhatók, melyek egy egyszerű metódushívással be is fejeződnek. Az adott bean példány csak az adott metódus végrehajtásának idejéig van klienshez rendelve, így az később újra kiosztható a kliensek között. Ilyen a példaalkalmazásban a rendelés elküldése folyamat.

A stateful session bean-ek kezelése a szerver számára sokkal nehezebb feladat, mint a stateless session bean-ek kezelése, mivel a bean állapotának eltárolása meglehetősen erőforrás-igényes feladat, így jelentősen a skálázhatóság rovására mehet használatuk. Ezzel szemben a stateless session bean-ek nagyon jól skálázhatók, mivel a bean állapotát a konténernek nem kell eltárolni a metódus hívások között.

A kliens a bean-nel az ún. üzleti interfészen keresztül lép kapcsolatba. Az üzleti interfész definiálja a bean kliens számára elérhető metódusait. Az interfész a bean metódusainak implementációját és a telepítési beállításokat elrejt a kliens elől. Ezeknek az interfészeknek két fajtájuk van:

- **Lokális interfész (locale interface):** A lokális kliensek (ugyanazon a gépen, és ugyanabban a JVM-ben futnak) tudnak kapcsolatba lépni a bean-nel ezen az interfészen keresztül. A lokális interfész metódusainak paraméterei a Java nyelvben megszokott módon, azaz referencia szerint adódnak át.
- **Távoli interfész (remote interface):** A távoli kliensek (más gépen futnak) ezen az interfészen keresztül tudnak kapcsolatba lépni a bean-nel. A referencia szerinti paraméterátadás nem járható út a távoli kliensek esetében, mivel ezek nem ugyanabban a JVM-ben futnak, mint a bean-ek. Távoli metódushívások mögött az RMI-IIOP protokoll húzódik meg. Ezen protokoll szerint a metódusok paraméterei referenciák helyett érték szerint adódnak át. Az érték szerinti paraméterátadásra a protokoll a Java nyelv objektum szerializációs mechanizmusát használja, ez azonban valamilyen szinten mindig a teljesítmény rovására megy.



5. ábra Távoli és lokális kliensek metódus hívásai

Lássunk tehát egy példát az alkalmazásból az állapotmentes session bean-re:

```
package cbs.ejbs;

import cbs.jpa_entities.Categories;
import java.util.List;
import javax.ejb.EJBException;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class CategoryEJBBean implements CategoryEJBLocal {

    @PersistenceContext
    private EntityManager em;

    public List<Categories> getCategories() {
        try {
            return (List<Categories>)
                em.createNamedQuery("Categories.findAll").getResultList();
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Mivel állapotmentes session bean-ről van szó, közvetlenül az osztálydefiníció elé a `@Stateless` annotációt helyezzük el. A bean a `CategoryEJBLocal` lokális üzleti interfészt implementálja. A bean-be DI-t használva beinjektáltam egy, a perzisztencia környezetnek megfelelő `EntityManager`-t. Látható az is, hogy a bean nem tartalmaz példányváltozót. A bean a lokális interfész egyetlen metódusát implementálja a `getCategories()` metódust, mellyel az alkalmazásban használt `Categories` entitások listáját adjuk vissza, felhasználva a beinjektált `EntityManager`-t és a `Categories` entitásnál létrehozott `Categories.findAll` nevesített lekérdezést, melyet a későbbiekben fogok jellemezni.

Nézzük még meg a CategoryEJBBean által implementált lokális interfészt:

```
package cbs.ejbs;

import cbs.jpa_entities.Categories;
import java.util.List;
import javax.ejb.Local;

@Local
public interface CategoryEJBLocal {

    public List<Categories> getCategories();

}
```

Lokális interfészről révén szó az interfészt a `@Local` annotációval látjuk el. Az interfész a lokális kliensek számára elérhető egyetlen metódust, a `getCategories()` metódust definiálja.

Most pedig következzen a már említett bevásárlókosár, melyet egy állapottal rendelkező session bean-nel valósítottam meg:

```
package cbs.ejbs;

import cbs.jpa_entities.Shoppingcarts;
import cbs.jpa_entities.ShoppingcartsPK;
import java.util.List;
import javax.ejb.EJBException;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateful
public class ShoppingcartEJBBean implements ShoppingcartEJBLocal {

    @PersistenceContext
    private EntityManager em;
    private String username;
    private List<Shoppingcarts> items;
    private float total;
```

```

public void init(String username) {
    if (username==null) {
        throw new EJBException("Username is null.");
    }
    this.username=username;
    items=this.getItems();
    total=0.0f;
    if (items!=null) {
        for(int i=0;i<items.size();i++) {
            total+=(items.get(i).getUnits()*items.get(i).getUnitPrice());
        }
    }
}

public float getTotal() {
    return total;
}

public void addItem(String isbn, int units, float unitPrice) {
    try {
        Shoppingcarts item=(Shoppingcarts)
            em.find(Shoppingcarts.class, new ShoppingcartsPK(username, isbn));
        if (item!=null) {
            em.remove(item);
            em.flush();
        }
        item=
            new Shoppingcarts(new ShoppingcartsPK(this.username, isbn), units, unitPrice);
        total+=(units*unitPrice);
        em.persist(item);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

public void removeItem(String isbn) {
    try {
        Shoppingcarts item=(Shoppingcarts)
            em.find(Shoppingcarts.class, new ShoppingcartsPK(username, isbn));
        if (item==null) {
            throw new Exception("The item is not found in the cart.");
        }
        total-=(item.getUnits()*item.getUnitPrice());
        em.remove(item);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

```

```

public void updateItemUnits(String isbn, int units) {
    try {
        Shoppingcart item=(Shoppingcart)
            em.find(Shoppingcart.class, new ShoppingcartPK(username, isbn));
        if (item==null) {
            throw new Exception("The item is not found in the cart.");
        }
        total+=((units-item.getUnits())*item.getUnitPrice());
        item.setUnits(units);
        em.persist(item);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

public List<Shoppingcart> getItems() {
    try {
        items=(List<Shoppingcart>)em.createNamedQuery(
            "Shoppingcart.findByUsername").setParameter("username", username).getResultList();
        em.clear();
        return items;
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

@Remove
public void deleteItems() {
    try {
        em.createQuery("DELETE FROM Shoppingcart s WHERE s.username=:username")
            .setParameter("username", username).executeUpdate();
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

@Remove
public void delete() {}
}

```

Látható, hogy a ShoppingcartEJBBean bean-t a @Stateful annotációval láttam el, jelezve, hogy állapottal rendelkező session bean. A bean a ShoppingcartEJBLocal lokális üzleti interfészt implementálja. Ebbe a bean-be is beinjektáltam egy EntityManager interfészt, mivel a bean entitásokkal fog műveletet végezni.

A bean a username, items és total példányváltozókkal rendelkezik. Ezek a változók rendre a felhasználó nevét, a bevásárlókosárban található elemeket és a bevásárlókosárban található elemek összesített árát tartalmazza.

Az `init()` metódust a bean létrehozása után fogja meghívni a kliens. Ezzel adjuk meg a felhasználó azonosítóját, és a kosár elemeire és összértéke is itt adunk meg kezdeti beállításokat. A `getTotal()` metódussal a kosár összértéke kérhető le. Majd az `addItem()`, `removeItem()`, `getItems()`, `deleteItems()` metódusok implementációja következik, melyekkel rendre hozzáadhatunk, eltávolíthatunk egy elemet a kosárból, lekérhetjük, illetve törölhetjük a kosár tartalmát. Az `updateItemUnits()` metódussal pedig a megadott könyvek példányszámát módosíthatjuk a kosárban.

A `@Remove` annotációval ellátott metódusok lefutása után a stateful session bean-t a konténer automatikusan eltávolítja.

A `ShoppingcartEJBBean` által implementált lokális üzleti interfész forráskódja:

```
package cbs.ejbs;

import cbs.jpa_entities.Shoppingcarts;
import java.util.List;
import javax.ejb.Local;
import javax.ejb.Remove;

@Local
public interface ShoppingcartEJBLocal {

    public void init(String username);

    public float getTotal();

    public void addItem(String isbn, int units, float unitPrice);

    public void removeItem(String isbn);

    public void updateItemUnits(String isbn, int units);

    public List<Shoppingcarts> getItems();

    @Remove
    public void deleteItems();

    @Remove
    public void delete();

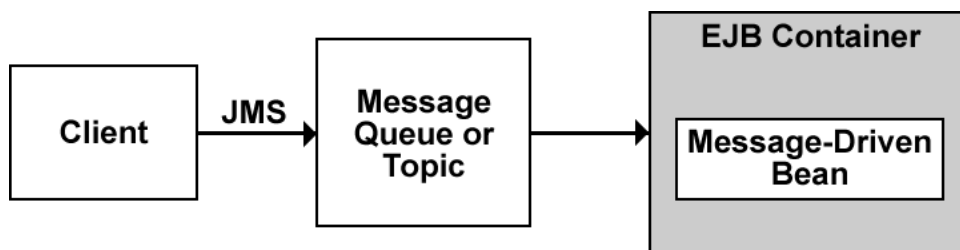
}
```

4.2. EJB üzenet-vezérelt bean-ek

Az üzenet-vezérelt bean lehetővé teszi üzenetek aszinkron feldolgozását [2] [6] [7]. Az üzenet-vezérelt bean működése leginkább az akció-figyelők működéséhez hasonlítható, annyi eltéréssel, hogy az MDB nem akciókat kap, hanem üzeneteket. Habár az MDB bármilyen üzenetet képes fogadni, de legtöbbször a Java Message Service API üzeneteit használják.

A session bean-ek szinkron működésűek. Ez azt jelenti, hogy amikor a kliens egy session bean-t hív, hogy valamilyen tevékenységet elvégezzon, addig a kliens várakozik. Amikor a bean elvégezte a műveletet, a vezérlés visszakérül a klienshez, mely folytatja működését. Számos alkalmazás esetében, mint például a business-to-business (B2B) alkalmazások, az aszinkron működés sokkal megfelelőbb.

A kliens soha sem hívja közvetlenül az üzenet-vezérelt bean metódusait. Ehelyett az üzenet-vezérelt bean-eket olyan üzenetek hozzák működésbe, melyeket egy üzenet kezelő szervernek küldünk, ami lehetővé teszi az üzenetek aszinkron küldését a rendszer komponensei között (6. ábra). Ezek az üzenetek származhatnak külső forrásokból, például egy másik, függetlenvállalati alkalmazás, vagy lehetnek belsők, azaz ugyanannak az alkalmazásnak más komponenseitől származnak. Az EJB konténernek köszönhetően, mely ezeket az üzeneteket az MDB-hez közvetíti, nem kell bonyolult infrastruktúrát vagy kezelő objektumokat fejlesztenünk ezen szolgáltatások eléréséhez.



6. ábra A kliens egy JMS üzenetet küld, melyet az MDB kap meg

Az MDB-nek elegendő egyetlen metódust, az `onMessage()` metódust definiálnia, mellyel az érkező üzeneteket kezeljük. Az üzenet-vezérelt bean-ek által biztosított funkcionalitás alapjaiban véve állapotmentes, hasonlóan a stateless session bean-ekhez. Az MDB nem tárol adatokat egyik üzenet feldolgozásától a másikig. Ha valamilyen az előző üzenet feldolgozása közben létrehozott adatokhoz szeretnénk hozzáférni, akkor azokat az adatokat valahol a bean-en kívül kell tárolnunk.

4.3. A Java Persistence API és a JPA entitások

A perzisztencia segítségével a Java objektumok által tartalmazott adatokat automatikusan relációs adatbázisokban tudjuk tárolni [10] [11]. A JPA az adatok automatikus kimentésének megvalósítására az ún. objektum-relációs leképezés (object-relational mapping-ORM) technikát használja. Az ORM lehetővé teszi, hogy konfigurálással a Java objektumokban tárolt adatokat relációs adatbázis táblázatokban tárolt adatoknak feleltessük meg, megkímélve a fejlesztőt az unalmas, alacsony szintű és sokszor komplex JDBC kódok írásától. A perzisztencia szolgáltató egy olyan ORM keretrendszer, mely támogatja a Java Persistence API-t.

Az API három fő részre osztható:

- ORM konfigurációs meta-adatok: Mint korábban már említettem, lehetővé teszik az entitások relációs adatbázis táblákhoz való rendelését. Az entitások azok a Java objektumok, melyeket adatbázisban tárolunk vagy úgy is fogalmazhatnánk, hogy az alkalmazás adatbázisban tárolt adatainak objektum-orientált reprezentációja. A JPA entitások a reláció és az objektum-orientált paradigma széles skáláját teszik elérhetővé egyszerre, mint az entitások közötti kapcsolatok, elsődleges kulcsok, az öröklődés vagy a polimorfizmus.
- Az EntityManager API: Egy olyan standard API, mellyel egyszerűen elvégezhetők az entitások ún. CRUD (create, read, update, delete) műveletei és egyéb perzisztencia műveletek. Az entitások a JPA szolgáltató felé elérhetővé teszik, hogy hogyan vannak az adatbázisra leképezve. Az EntityManager interfész ezeket az ORM meta-adatokat olvasva hajtja végre a perisztencia műveleteket. Tulajdonképpen az EntityManager tudja, hogy hogyan kell az entitásokat az adatbázishoz hozzáadni, a tárolt entitásokat elérni és frissíteni és hogy hogyan törölje az entitásokat az adatbázisból. Emellett a JPA az EntityManager interfészen keresztül biztosítja a lehetőséget az életciklus kezeléshez, a teljesítmény növeléshez, a gyorsítótárhoz, és a tranzakció kezeléshez is.
- A Java Persistence Query Language (JPQL): A JPA egy speciális, az SQL-hez hasonló lekérdező nyelvet is biztosít az adatbázisban tárolt entitások eléréséhez.

Az példaalkalmazás relációs adatbázisát létrehozó SQL utasítások, mely adatbázisra a JPA entitások épülnek, megtalálhatók a Függelék A részében.

Az entitás osztályok létrehozásának jellegzetességeit a példaalkalmazás egyik JPA entitásával, a Categories entitással fogom bemutatni, melynek forráskódja az alábbi:

```

package cbs.jpa_entities;

import java.io.Serializable;
import java.util.List;
import javax.persistence.*;

@Entity
@Table(name = "CATEGORIES")
@NamedQueries({
    @NamedQuery(
        name = "Categories.findAll",
        query = "SELECT c FROM Categories c"),
    @NamedQuery(
        name = "Categories.findById",
        query = "SELECT c FROM Categories c WHERE c.categoryId = :categoryId"),
    @NamedQuery(
        name = "Categories.findByName",
        query = "SELECT c FROM Categories c WHERE c.categoryName = :categoryName")
})
public class Categories implements Serializable {

    private static final long serialVersionUID = 1L;
    @TableGenerator(
        name="PK_CATEGORIES_TABLE_GEN",
        table="PK_TABLE",
        pkColumnName="PK_NAME",
        valueColumnName="PK_VALUE",
        pkColumnValue="CATEGORIES_PK",
        allocationSize=1
    )
    @Id
    @Basic(optional = false)
    @Column(name = "CATEGORY_ID")
    @GeneratedValue(generator="PK_CATEGORIES_TABLE_GEN",strategy=GenerationType.TABLE)
    private Integer categoryId;
    @Basic(optional = false)
    @Column(name = "CATEGORY_NAME")
    private String categoryName;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "categoryId")
    private List<Books> booksCollection;

    public Categories() {
    }

    public Categories(Integer categoryId) {
        this.categoryId = categoryId;
    }

    public Categories(Integer categoryId, String categoryName) {

```

```

        this.categoryId = categoryId;
        this.categoryName = categoryName;
    }

    public Integer getCategoryId() {
        return categoryId;
    }

    public void setCategoryId(Integer categoryId) {
        this.categoryId = categoryId;
    }

    public String getCategoryName() {
        return categoryName;
    }

    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }

    public List<Books> getBooksCollection() {
        return booksCollection;
    }

    public void setBooksCollection(List<Books> booksCollection) {
        this.booksCollection = booksCollection;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (categoryId != null ? categoryId.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Categories)) {
            return false;
        }
        Categories other = (Categories) object;
        if ((this.categoryId == null && other.categoryId != null) ||
            (this.categoryId != null && !this.categoryId.equals(other.categoryId))) {
            return false;
        }
        return true;
    }
}

```

```

@Override
public String toString() {
    return "cbs.jpa_entities.Categories[categoryId=" + categoryId + "];
}
}

```

Az osztály a szokásos JavaBeans szabályokat követi. A példányváltozók nem publikusak, kliensek a megfelelő `getXXX()` és `setXXX()` metódusokon keresztül érhetik el azokat.

A hagyományos POJO-tól csak az annotációk különböztetik meg, melyek az entitás meta-adatait írják le, azokat az adatokat, hogy az entitást hogyan tároljuk, és hogy milyen kapcsolatban áll más entitásokkal.

Az `@Entity` annotációval jelezzük a perzisztencia motornak, hogy a `Categories` osztály egy entitás. Az annotációt közvetlenül az osztálydefiníció elé helyezzük el, jó példa az osztály-szintű annotációra.

Az entitás implementálja a `Serializable` interfészt, melyre akkor van szükségünk, ha az entitást távoli kliensnek továbbítjuk, ilyenkor az érték szerinti paraméterátadás miatt az objektumot a Java szerializációs mechanizmusa adja át érték szerint.

Az osztálynak kötelezően rendelkeznie kell egy publikus (`public`) vagy védett (`protected`) konstruktorral, emellett még két másik konstruktora is van.

Az osztály nem deklarálható véglegesként (`final`), emellett a metódusok és a perzisztens példányváltozók sem láthatók el a `final` kulcsszóval.

A `@Table` annotációval azt adhatjuk meg, hogy az adott entitás osztályt, az adott perzisztencia környezet mely adattáblájához rendeljük hozzá. Jelen esetben a `Categories` entitást az adatbázis `CATEGORIES` táblájához rendeljük hozzá. A `@Column` annotációval pedig az egyes példányváltozókat rendelhetjük az adattábla oszlopaihoz.

Az `@Id` annotációval jelöljük, hogy a `categoryId` példányváltozó elsődleges kulcsként funkcionál. Hasonlóan a relációs modellhez, az elsődleges kulcs alapján tudjuk az entitásokat egymástól egyértelműen megkülönböztetni.

A kategóriák esetén automatikus elsődleges kulcs generálást alkalmaztam. Ez azt jelenti, hogy amikor egy új kategóriát hozunk létre, akkor az elsődleges kulcs mezőt (a `categoryId` mezőt) nem kell explicit módon megadni, hanem azt különböző kulcsgenerálási stratégiák szerint a perzisztencia szolgáltató automatikusan elvégzi. Ilyen stratégia az általam alkalmazott `TABLE` stratégia, melynek lényege, hogy az elsődleges kulcsok neveit és aktuális

értéküket az adatbázis egy táblájában tároljuk, és minden esetben amikor az entitás egy újabb példányát rögzítjük az adatbázisban, akkor a kulcs aktuális értékét növeljük. Ezeket a kulcsgenerálási beállításokat a `@TableGenerator` és `@GeneratedValue` annotációk segítségével lehet egyszerűen elvégezni.

Ahogy a relációs modellben az adattáblákat kapcsolatokon keresztül kapcsoljuk össze egymással, úgy ez igaz az entitásokra is. Az egyes kapcsolatfajtákhoz a JPA biztosítja a megfelelő kapcsolatot modellező annotációkat. A kapcsolat aszerint, hogy a kapcsolatban részt vevő entitások közül csak az egyik, vagy mindkettő rendelkezik kapcsolat mezővel, lehet egyirányú, vagy kétirányú. A `Categories` osztály tartalmaz egy `booksCollection` példányváltozót a `Books` osztállyal való kapcsolat leírására. Ez a mező a `@OneToMany` annotációval van ellátva, mellyel jelezzük, hogy a kategóriák egy-a-többhöz kapcsolatban állnak a könyvekkel és a kapcsolat kétirányú.

Az osztálydeklaráció előtt található egy `@NamedQueries` annotáció, mellyel három nevesített lekérdezést hoztam létre. Ezek a nevesített lekérdezések lehetővé teszik, hogy különböző JPQL lekérdezéseket névvel lássunk el és a későbbiekben ezen a néven hivatkozhatunk rájuk. Főleg bonyolultabb és gyakran használt lekérdezések esetében hasznosak.

5. A Java Servlet Technológia

A Servlet olyan dinamikus tartalom generálására használható Java osztály, mely lehetővé teszi a kérés/válasz alapon működő szerver nyújtotta szolgáltatások kiterjesztését [3] [11]. Eredetileg a Servlet bármilyen típusú kérést (például HTTP, SMTP, FTP kérések) tud kezelni, azonban a gyakorlatban a HTTP kérések kezelésére írt Servletek terjedtek el.

5.1. A Servletek és a hagyományos CGI programok összehasonlítása

A Common Gateway Interface (CGI) volt az egyik első dinamikus tartalom készítésére használt, széles körben elterjedt technológia. A CGI segítségével a Web-szerver bizonyos kéréseket egy külső programnak ad át további feldolgozásra, majd ennek a programnak a kimenete rendszerint egy statikus fájl formájában kerül vissza a klienshez. A CGI megjelenése új funkciók egész sorát tette elérhetővé a Web oldalakban, ennek köszönhetően gyorsan elterjedt, de több hátrányos tulajdonsága is van.

A CGI egyik legfőbb hátránya, hogy minden egyes HTTP kérés kezeléséhez új folyamatot kell a szervernek elindítania, ami rendkívül idő- és erőforrás igényes feladat és erősen limitálja, hogy a szerver hány kérést tud kezelni egyszerre. Ezzel szemben a HTTP kérések kezelésére írt Servlet osztálynak csak egyetlen példánya létezik a JVM-ben, és a kérések kezelésére csak egy könnyűsúlyú Java szál indul el, így jobban takarékoskodik az erőforrásokkal.

A Servletek több olyan lehetőséget is támogatnak, melyek megvalósítása CGI-vel rendkívül nehézkes vagy lehetetlen. Például a Servlet közvetlenül tud kommunikálni a Web-szerverrel, míg a hagyományos CGI programok nem, vagy csak szerver-specifikus API-k segítségével, ami a hordozhatóság rovására megy. A Servlet képes tárolni adatokat a kérések között, így egyszerűsíthetők olyan technikák, mint a session tracking (munkaviszony követés).

A Servleteket a Java programozási nyelven írják és standardizált API-t követnek, így hordozhatók és szinte az összes fontosabb Web-szerver támogatja őket.

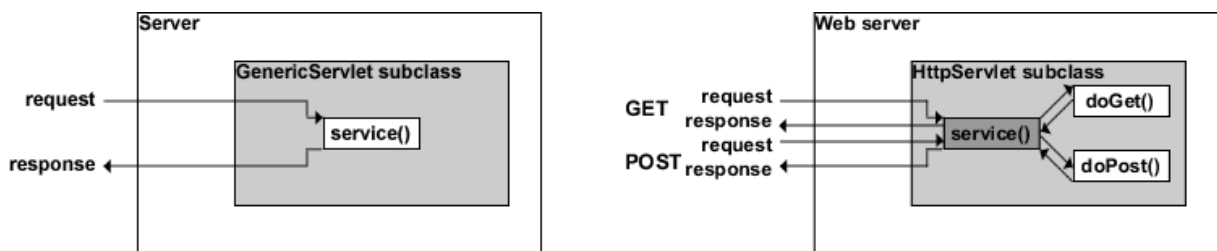
A Servletek biztonságosabbak is, mint a CGI programok, köszönhetően a Java nyelvben megtalálható automatikus biztonsági mechanizmusoknak, mint például a tömbhatár figyelés, vagy az automatikus szemétgyűjtés.

5.2. A Servlet API

A Servletek alapjaiban véve két Java csomag, a `javax.servlet` és a `javax.servlet.http`, osztályait és interfészeit használják. A `javax.servlet` csomag tartalmazza az általános célú Servletek készítését támogató osztályokat és interfészeket. Ezen osztályokat és interfészeket terjeszti ki a `javax.servlet.http` csomag a HTTP-specifikus Servletek készítéséhez.

Minden Servlet implementálja a `javax.servlet.Servlet` interfészt. Általános, protokoll-független szolgáltatás implementálásához a `javax.servlet.GenericServlet` osztályt szokás kiterjeszteni és a `service()` metódusát felülírni.

HTTP-specifikus szolgáltatás implementálásához a `javax.servlet.http.HttpServlet` osztályt titerjesztjük ki és nem a `service()` metódust írjuk felül, hanem a HTTP kérés típusának megfelelő `doXXX()` metódust, például GET kéréshez a `doGet()` metódust, POST kérés esetén a `doPost()` metódust (7. ábra). A `service()` metódust azért nem szokás felülírni, mivel ez a metódus felelős a kérés típusának megfelelő `doXXX()` metódus paramétereinek beállításáért és a megfelelő metódus meghívásáért.

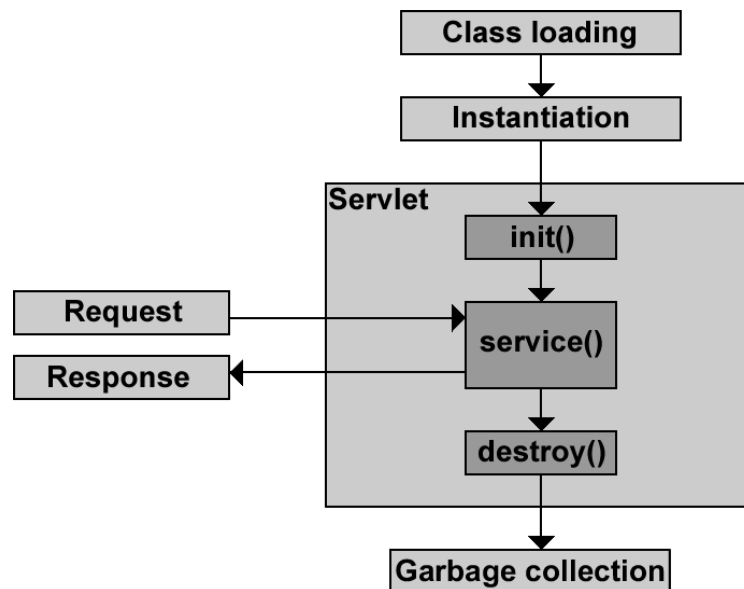


7. ábra Egy általános és egy HTTP Servlet kérés kezelése

A csomagok több támogató osztályt és interfészt is tartalmaznak. Például a `javax.servlet` csomagban található `ServletRequest` és `ServletResponse` osztályok lehetővé teszik a hozzáférést az általános kérésekhez és válaszokhoz, míg a `javax.servlet.http` csomagban található `HttpServletRequest` és `HttpServletResponse` osztályok a HTTP kérésekhez és válaszokhoz biztosítanak hozzáférést. Ezekon kívül a `javax.servlet.http` csomag tartalmaz egy `HttpSession` interfészt is, ami beépített session tracking funkcionalitást biztosít és egy `Cookie` osztályt, mely HTTP cookie-k (sütek) beállításához és feldolgozásához ad segítséget.

A Servleteket a Web konténer kezeli. A Web konténer a következő lépéseket hajtja végre, amikor egy kérés érkezik a Servlethez:

- Ha a Servletnek nem létezik példánya, akkor a konténer
 - Betölti a Servlet osztályt,
 - Példányosítja a Servletet az argumentum nélküli konstruktor meghívásával,
 - Meghívja az `init()` metódust, mellyel inicializálja a Servlet példányt
- Meghívja a `service()` metódust, melynek átadja paraméterként a Request és Response objektumokat. Ha a Servlet egy `HttpServlet`, akkor `service()` metódus a HTTP kérés típusának megfelelő `doXXX()` metódust hívja meg.
- Ha a konténernek valamilyen okból el kell távolítania a Servletet, akkor meghívja a `destroy()` metódust.



8. ábra A Servlet életciklus

Míg a Servlet életciklusa során az `init()` és a `destroy()` metódus egyszer hajtódik végre, addig a `service()` metódus többször.

A Web konténer az `init()` metódust közvetlenül a példányosítás után hívja meg, még mielőtt a Servlet bármely kérést kezelne. Olyan beállítások végrehajtását, erőforrások lefoglalását helyezzzük ide, ami a szolgáltatás nyújtásához szükséges, például adatbázis kapcsolatok létrehozása.

Minden egyes kéréshez a szerver elindít egy új szálát és meghívja a `service()` metódust, mely a válasz generálásáért felelős. Két paramétere van, melyek a kienstől érkező kérést és a kliensnek szóló választ reprezentálják.

A `destroy()` metódust akkor hívja meg a Web konténer, amikor befejeződik a szolgáltatás nyújtása és az összes függő kérés sikeresen vagy időtúllépés miatt véget ér. Itt a Servlet által használt erőforrásokat szabadíthatjuk fel.

5.3. A HelloWorldServlet

A példaalkalmazás nem tartalmaz általam készített Servletet, ezért egy külön írt Servleten keresztül mutatom be az alapvető Servlet koncepció lényegesebb elemeit, melynek forráskódja az alábbi:

```
package webtestapp;

import java.io.*;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.http.*;

public class HelloWorldServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet HelloWorldServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Hello World!</h1>");
            out.println("The actual date is: "+new Date());
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
}
```

A `HelloWorldServlet` a `HttpServlet` osztályt terjeszti ki, mivel HTML tartalmat fogunk vele generálni. A Servlet egy metódust, a `doGet()` metódust definiálja. Ez a metódus fog lefutni minden olyan esetben, amikor egy olyan URL-hez érkezik HTTP GET kérés, mely

ehhez a Servlethez van rendelve. A `doGet()` metódust a `HttpServlet` alapértelmezett `service()` metódusa hívja meg és két objektumot ad át neki paraméterként. Az egyik egy `HttpServletRequest` objektum, mely az aktuális kérésről tartalmaz információkat, a másik egy `HttpServletResponse` objektum, mely a válasz generálásához biztosítja a megfelelő mechanizmust.

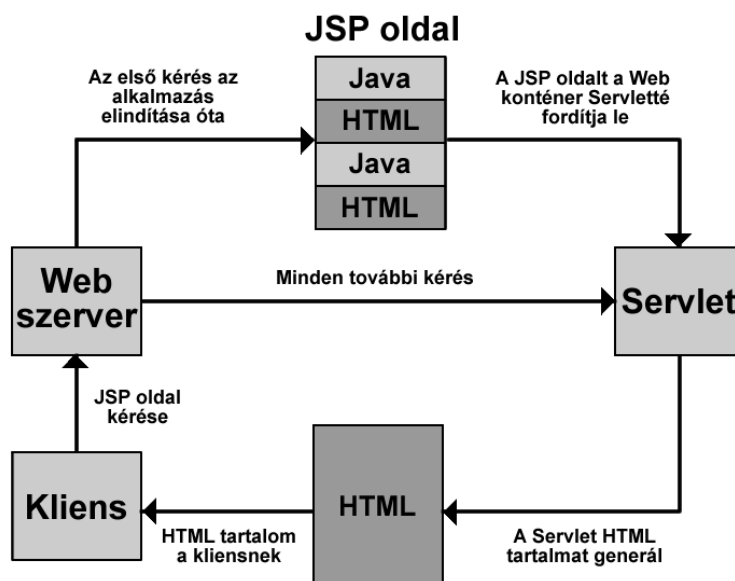
A `doGet()` metódus tulajdonképpen három dolgot csinál. Először is, beállítja a válasz típusát `text/html` típusra, mellyel jelezzük, hogy a Servlet standard HTML szöveget fog válaszként létrehozni. Ezután a `HttpServletResponse` paraméter `getWriter()` metódusát hívjuk meg, hogy a kliensre mutató `java.io.PrintWriter` objektumot kapjunk. Végül ezt az objektumot használva a HTML tartalmat visszaküldjük a kliensnek. Látható hogy a statikus tartalomba ágyaztunk némi dinamikus tartalmat is egy `java.util.Date` objektumon keresztül, mely az aktuális dátumot és időt reprezentálja.

A példaalkalmazásban mégis található egy Servlet a `FacesServlet`, melyet a JSF technológiát bemutató részben fogok ismertetni.

6. A JavaServer Pages (JSP) technológia

Bár a Java Servlet technológia az alapja a szerver-oldali Java programozásnak, a fejlesztési idő tekintetében nem a leghatékonyabb, mivel a Servlet legkisebb változtatása is a forráskód módosításával, újrafordításával és újratelepítésével jár. Ráadásul a Webes tartalom generálása főként az `out.println()` utasításokkal történik, mely rendkívül körülményessé és nehézkesé teszi a fejlesztést és a hibakeresést. Ezen hátrányok orvosolására és az olyan technológiák versenytársaként, mint az ASP, alkották meg a JSP technológiát, mely Java kód beágyazást teszi lehetővé statikus szövegbe.

A JSP dinamikus Webes tartalom generálására szolgáló technológia [4] [11]. A JSP oldal olyan szöveges dokumentum, mely két fajta szöveget tartalmaz: statikus adatot, mely lehet bármilyen szöveg-alapú formátumban (pl. HTML, XML, XHTML), és JSP elemeket, melyek a dinamikus tartalom generálásáért felelnek.



9. ábra A JSP életciklus

A JSP a Java Servlet technológiára épül, a Web konténer automatikusan minden JSP oldalt konkrét Servletté fordít le. Ebből következően a JSP elérhetővé teszi a Servlet technológia dinamikus képességeit, de egy sokkal természetesebb megközelítést alkalmaz a Webes tartalom készítéséhez. Amikor a Web szerverhez olyan kérés érkezik, mely egy JSP oldalra mutat, akkor a kérést a szerver a Web konténernek továbbítja. Ha a JSP nincs még lefordítva, akkor a konténer Servletté fordítja le a JSP oldalt, betölti a Servletet és a konténer

továbbítja felé a kérést. A Servlet feldolgozza a kérést, és visszaküldi a választ a kliens felé. Az összes többi kérés közvetlenül a Servlet felé irányítódik és a klasszikus Servlet életciklus megy végbe.

6.1. JSP jelölő elemek

Valahogyan azonban jelezni kell a fordító felé, hogy az oldalnak mely részei Java kódok és mely részei hagyományos HTML szövegek. Erre a Java specifikáció különböző jelölő elemeket (tag-eket) definiál, melyek egyértelműen magukba foglalják a Java kódot. Ezek a jelölő elemek kétféle szintaktikát követhetnek, használhatók a hagyományos HTML-szerű jelölő elemek, és létezik ezen elemek XML szintaktikát követő párai is, de ha döntöttünk valamelyik szintaktika mellett, akkor konzekvensen azt kell alkalmaznunk az adott JSP oldalon belül. A példaalkalmazásban következetesen a HTML-szerű szintaktikát használom.

Ezeket a jelölő elemeket alapvetően három fő kategóriába lehet sorolni aszerint, hogy milyen funkciót töltenek be:

- Script elemek
- Akció elemek
- Direktívák

A script elemek lehetővé teszik, hogy programkódot ágyazzunk be közvetlenül a JSP oldalba. Annak ellenére, hogy a JSP specifikáció lehetővé teszi, hogy nemcsak Java kódot ágyazzunk be a JSP oldalba, csak néhány nagyon speciális konténer rendelkezik ilyen képességekkel, így a beágyazott kód szinte mindig Java kód. A script elemek három típusa a scriptlet, a kifejezés és a deklaráció.

Az akció elemek, olyan jelölők, melyek közvetlenül részt vesznek a kérés feldolgozásában. Ezek a jelölő elemek lehetővé teszik, hogy hozzáférjünk adatokhoz és ezeket az adatokat módosítsuk vagy bizonyos transzformációkat hajtsunk végre rajtuk a dinamikus tartalom generálása céljából. Az akció elemek lehetnek standard elemek, melyek a JSP specifikáció határoz meg és lehetnek egyedi jelölő elemek, melyeket a JSP jelölő elem kiterjesztő mechanizmusa segítségével hozunk létre. Ez a mechanizmus lehetővé teszi, hogy a fejlesztők saját jelölő elemeket hozzanak létre az adatok manipulálására és a dinamikus tartalom generálására.

A gyakran használt funkciók egyszerű használata és szabványosítása céljából hozták létre a JavaServer Pages Standard Tag Library-t, röviden a JSTL-t. A JSTL a következő típusú jelölő elemeket tartalmazza:

- Alap funkciókat megvalósító elemek, mint például az iterációs és a logikai utasítás elemek.
- XML feldolgozó elemek
- Formázó elemek, melyek segítenek a lokalizált és többnyelvű alkalmazások egyszerűbb elkészítésében
- Adatbázis utasításokat tartalmazó jelölő elemek

A többi JSP jelölő elemmel ellentétben a direktívák közvetlenül nem használhatók dinamikus tartalom generálására, ehelyett a JSP oldal tulajdonságait és viselkedését irányíthatjuk velük. Olyan speciális utasításokkal is elláthatjuk a konténert, hogy hogyan járjon el az oldal fordítása közben. Három fajta direktíva létezik. A page direktíva a JSP oldal attribútumainak beállítás teszi lehetővé, az include direktíva segítségével másik JSP oldalt szűrhatunk be az oldalba, míg a taglib direktívával hivatkozhatunk azokra a könyvtárakra, melyeknek az elemeit a JSP oldalban felhasználtuk.

6.2. Az egységesített EL

A JSP 2.0-tól kezdve a specifikáció részét képezi az Expression Language (EL), azaz magyarul a kifejezés nyelv [11]. A fő ok, mely az EL bevezetése mellett szólt, hogy biztosítsák a script elemek nélküli JSP oldalak készítését, melyek a nagyobb projektek esetében bonyolulttá tették az oldalakat. A script elemek használatát a JSP oldalban az adott alkalmazás által támasztott követelmények határozzák meg. A főbb ilyen követelmények melyek elengedhetetlenné tették a script elemek használatát a következők voltak:

- Vezérlési folyamat biztosítása a JSP oldalnak
- A JSP oldal lokális változóinak beállítása és azok elérése
- Olyan komplex kifejezések értékének megjelenítése, melyek Java objektumokat is érintenek
- Tetszőleges Java objektum tulajdonságainak elérése
- JavaBeans vagy egyéb Java objektumok metódusainak hívása

Ezek közül az első kettőt a JSTL segítségével lehet kezelni, a többi három követelmény kezelésére pedig az EL használható. Az EL kifejezések két helyre helyezhetők el

a JSP oldalban, közvetlenül a statikus szövegbe, vagy a JSP akció elemek attribútumaiba. Az EL-ben található 11 ún. implicit objektum is a JSP programozás megkönnyítésére. Ezek az objektumok az alábbiak: a JSP implicit objektum, a érvényességi tartományoknak megfelelő objektumok, a paraméterek elérését biztosító implicit objektumok, a fejrészt elérő objektumok, és az inicializációs paramétereket elérő implicit objektum.

A JSP 2.1 legfőbb újítása az egységesített EL (unified EL), mely a JSP által használt kifejezés nyelv és a JSF kifejezés nyelvének az uniója. Összegezve az új, egységesített EL lehetővé teszi egyszerű kifejezések használatával, hogy

- Dinamikusan olvassunk ki JavaBeans objektumokból, különböző adatszerkezetekből és implicit objektumokból adatokat

- Dinamikusan írjunk ki adatokat JavaBeans komponensekbe
- Tetszőleges statikus és publikus metódus hívását
- Aritmetikai műveletek dinamikus végrehajtását

Az egységesített EL azt is megengedi az egyedi jelölő elemeket készítőknél, hogy meghatározzák, hogy az egyedi elem attribútumai milyen kifejezést fogadnak el az alábbiak közül:

- Azonnali kifejezés kiértékelés vagy késleltetett kifejezés kiértékelés
- Érték kifejezés vagy metódus kifejezés
- Jobbérték vagy balérték

7. A JavaServer Faces (JSF) Technológia

A JSF egy viszonylag új technológia, mellyel sokkal gazdagabb felhasználói felületeket készíthetünk a Webes alkalmazásoknak, mint tisztán a Servlet vagy a JSP technológia segítségével [5] [9] [11]. A JSF technológiát a Servlet és a JSP technológiákkal együtt, mint támogató technológiát kell értelmezni. A példaalkalmazásban én is előszeretettel támaszkodom a JSF technológia használatára.

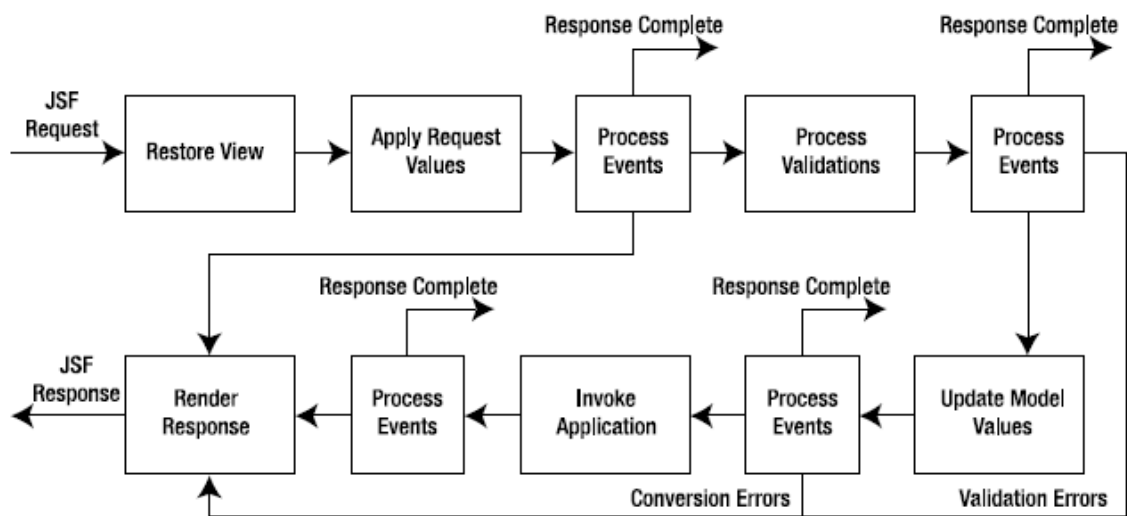
A JSF egy esemény-vezérelt, komponens-alapú modellt határoz meg a Webes alkalmazások számára, hasonlóan az önálló GUI alkalmazásokban évek óta sikeresen használt Modell-View-Controller (MVC) modellhez. Ezek a komponensek olyan felhasználói felület komponensek, melyekkel egyszerűen készíthetők, szerver-oldali felhasználó felületek. A JSF egyszerűvé teszi azt is, hogy ezeket a komponenseket az alkalmazás adatforrásaihoz kapcsoljuk, és hogy a kliens által generált eseményeket szerver-oldali esemény kezelők dolgozzák fel.

A JSF a következő módokon segíti a Webes alkalmazás fejlesztőket a felhasználói felület készítésében:

- Egyszerűvé teszi, hogy felhasználói felületeket készítsünk újrafelhasználható komponensek segítségével
- Egyszerűsíti a felhasználó felület és az alkalmazás adatainak egymáshoz rendelését
- Segít a felhasználó felület állapotának menedzselésében a különböző kérések között
- Olyan modellt biztosít, mellyel egyszerűen rendelhetők szerver-oldali esemény kezelőkhöz, a kliens által generált események
- Lehetővé teszi egyedi komponensek készítését és újrafelhasználását

A JSP-hez hasonlóan a JSF is rendelkezik élet ciklussal, mely hat jól meghatározott állapokra osztható:

- Nézet visszaállítása (Restore view)
- Kérési adatok alkalmazása (Apply request values)
- Adatok érvényesítése (Process validations)
- A modell adatainak frissítése (Update model values)
- Alkalmazás hívása (Invoke application)
- Válasz megjelenítése (Render response)



10. ábra A JSF élekciklus

A JSF technológia alapmodellje az MVC modellre épül. Az MVC modell lehetővé teszi, hogy különválasszuk a nézetet (jelen esetben az oldalakat) és a modellt (a nézet által megjelenített adatokat). Ebben a modellben a kontroller kezeli a felhasználói eseményeket, melyek hatására a modell adatai megváltozhatnak és frissíti a nézetet. A JSF modellben ez a kontroller egy Servlet, a FacesServlet. Minden kérés a FacesServlethez érkezik, mely megvizsgálja a kérést és különböző műveleteket hajt végre a modellben.

Ez a Servlet a JSF része, és csak külső meta-adatok révén konfigurálható, melyeket a faces-config.xml konfigurációs fájlban kell megadnunk, mely a telepítés leíró fájlhoz hasonlít. Ebben a fájlban például az alkalmazásra vonatkozó információkat, navigációs utasításokat, egyedi komponenseket, adatérvényesítést és adatkonvertálásért felelős osztályokat adhatunk meg. Ebből a konfigurációs fájlból lentebb megadok egy részletet, melyben az alkalmazáshoz rendelünk egy a megadott e-mail cím helyességét ellenőrző adatérvényesítőt, egy menedzselt JavaBeans komponenst, mely a regisztrációs folyamatban játszik szerepet és egy navigációs szabályt, mellyel azt definiáljuk, hogy ha a registration.jsp regisztrációs oldalon sikeres regisztrációs visszajelzést kapunk, akkor irányítson át a login.jsp bejelentkezés oldalra:

```

<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="1.2"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
  <validator>
    <validator-id>emailValidator</validator-id>
    <validator-class>cbs.validators.EmailValidator</validator-class>
  </validator>
  <managed-bean>
    <managed-bean-name>regBean</managed-bean-name>
    <managed-bean-class>cbs.managed_beans.RegistrationJSFBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/registration.jsp</from-view-id>
    <navigation-case>
      <from-outcome>reg_success</from-outcome>
      <to-view-id>/login.jsp</to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>

  ....további elemek
</faces-config>

```

Nézzünk meg a példaalkalmazás prezentációs rétegéből egy JSP oldalt, a login.jsp bejelentkezés oldalt:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<f:view locale="#{settingsBean.language}">
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <link rel="stylesheet" type="text/css" href="faces/WEB-INF/styles/style_default.css">
    <title><h:outputText value="#{msgs.login_title}"/></title>
  </head>
  <body>
    <h:panelGrid styleClass="container">
      <h:form>
        <%@include file="WEB-INF/jspf/header.jspf"%>
      </h:form>
      <h:panelGrid styleClass="title">
        <h:panelGroup>
          <h1><h:outputText value="#{msgs.login_title}"/></h1>
        </h:panelGroup>
      </h:panelGrid>
      <h:panelGrid styleClass="content">
        <h:outputText value="#{msgs.login_message}"/>
        <%@include file="WEB-INF/jspf/login.jspf"%>
      </h:panelGrid>
      <%@include file="WEB-INF/jspf/footer.jspf"%>
    </h:panelGrid>
  </body>
</html>
</f:view>
```

Látható, hogy a standard HTML kód mellett más elemek is találhatók a forráskódban. Rögtön az oldal elején található néhány JSP jelölő elem. A `<%@page ...%>` direktíva segítségével az oldal tartalmának típusát és karakterkódolását állítjuk be. Emellett más, az oldalra vonatkozó beállítások is megadhatók itt.

A `<%@taglib ...%>` direktívával adjuk meg azokat az egyedi elem könyvtárakat, melyeket az oldalban felhasználunk. Ezeket a könyvtárakat az oldalra nézve egyedi

prefixekkel kell ellátnunk. Az oldal két ilyen könyvtárat használ a JSF egyedi könyvtáraiból, a core és html könyvtárat, melyeknek konvencionálisan az f és h prefixeket adtam.

Minden JSF elemet az `<f:view> ... </f:view>` elemek közé kell elhelyeznünk. Látható, hogy a view elemen belül a HTML, a JSP és a JSF elemek keverten fordulnak elő. A view elem locale attribútumához megadott érték egy EL kifejezés, a settingsBean menedzselte bean language mezőjét adja vissza, mellyel a felhasználó által preferált nyelvre állítja be az oldalt. A `<h:outputText .../>` JSF elem például szöveg megjelenítésére használható. A value attribútumában egy EL kifejezés található, mellyel lokalizált szöveget rendelünk a komponenshez.

Egy másik JSP direktíva az `<%@include ...%>` direktíva, mely segítségével JSP részleteket adhatunk hozzá az aktuális oldalhoz. Az include direktíva akkor kerül feldolgozásra, amikor a JSP oldalt a konténer Servlett fordít.

Lássunk egy ilyen JSP részletet, a login.jspf fájlt:

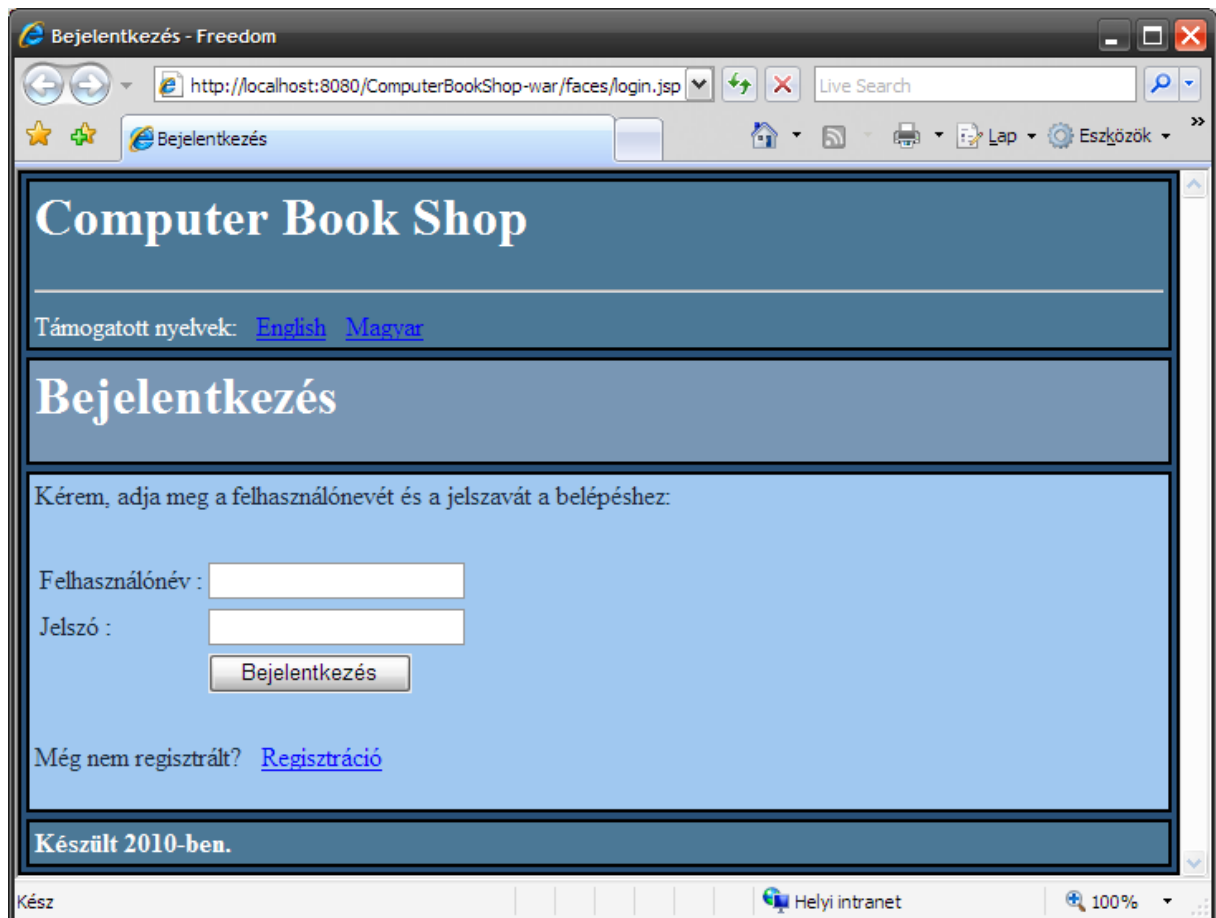
```
<%@page pageEncoding="UTF-8"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<form action="j_security_check" method="post">
    <br>
    <h:panelGrid columns="2">
        <h:panelGroup>
            <h:outputText value="#{msgs.include_login_username}"/>
        </h:panelGroup>
        <h:inputText id="j_username"/>
    </h:panelGroup>
    <h:panelGroup>
        <h:outputText value="#{msgs.include_login_password}"/>
    </h:panelGroup>
    <h:inputSecret id="j_password"/>
    <h:panelGroup/>
    <h:commandButton value="#{msgs.include_login_button}"/>
    </h:panelGrid>
</form>
<h:form>
    <br>
    <h:outputText value="#{msgs.include_login_reg_question}"/>
    &nbsp;
    <h:commandLink action="go_to_reg_page" value="#{msgs.include_login_reg}"/>
</h:form>
```

Konvencionálisan a JSP részleteket .jspxf kiterjesztéssel látjuk el és a Web modul WEB-INF/jspf könyvtárában helyezzük el őket.

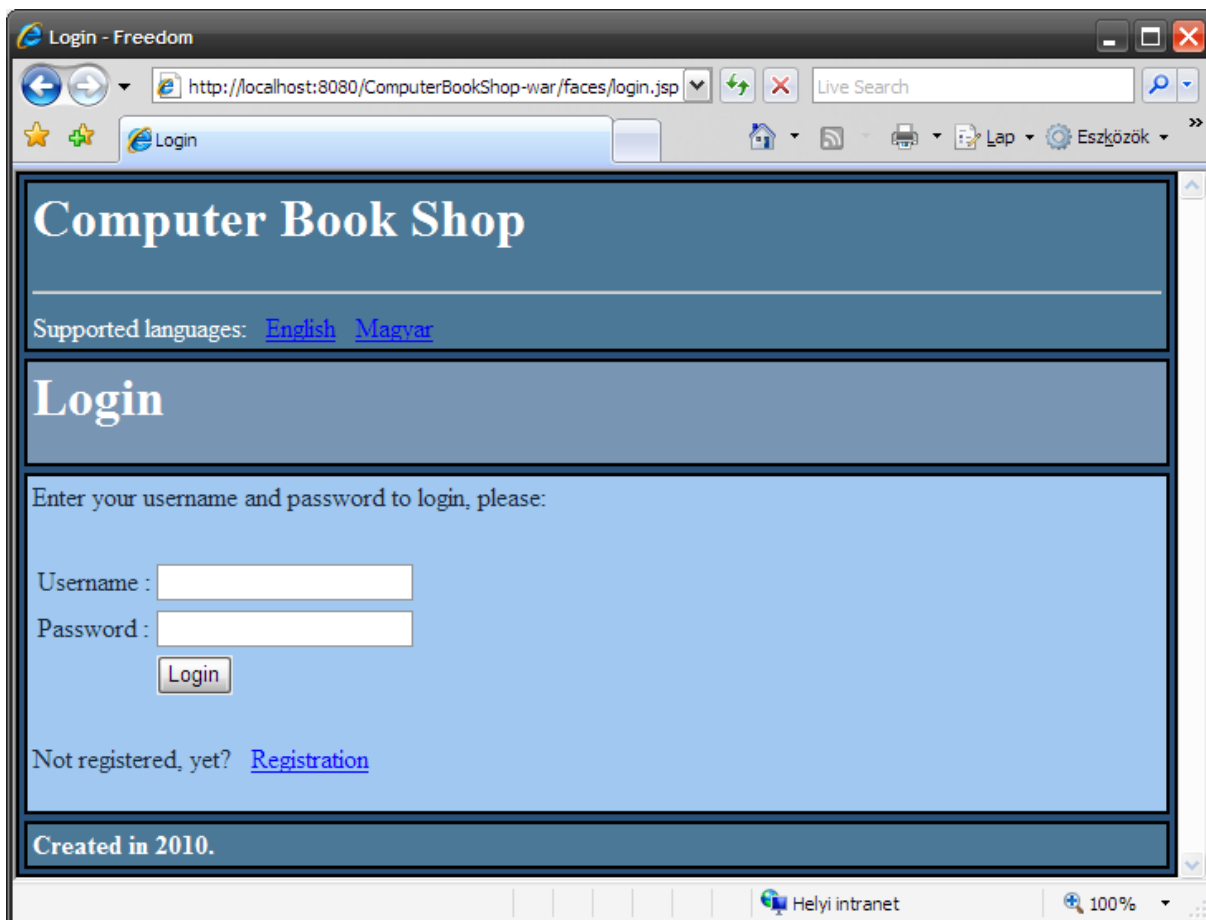
A JSP részlet két direktívával kezdődik, majd egy HTML form elemmel folytatódik. Érdekes lehet a form action paramétereként megadott `j_security_check` érték, és a két JSF inputmező azonosítója a `j_username` és a `j_password`. Ezekkel az értékekkel jelezzük, hogy a Java Authentication and Authorization Service (JAAS) form-alapú hitelesítő mechanizmusát használjuk. A `<h:commandLink .../>` elemmel egy linket hozunk létre, action attribútumának értékét a JSF konfigurációs fájlban használjuk fel az oldalak közötti navigálásra.

Nézzük meg hogyan jeleníti meg a böngésző a fent bemutatott `login.jsp` oldalt:



11. ábra A `login.jsp` bejelentkezés oldal magyar nyelven

Most ugyanez az oldal angol nyelven:



12. ábra A login.jsp bejelentkezés oldal angol nyelven

A JSF technológiában a prezentációs tartalmat az üzleti réteggel az ún. menedzselt bean-eken keresztül kapcsoljuk össze. Ilyen menedzselt bean a stockBean, amely a StockJSFBean osztály példánya:

```
package cbs.managed_beans;

import cbs.ejbs.BookEJBLocal;
import cbs.ejbs.CategoryEJBLocal;
import cbs.jpa_entities.Books;
import cbs.jpa_entities.Categories;
import java.util.List;
import java.util.Map;
import javax.ejb.EJB;
import javax.faces.context.FacesContext;

public class StockJSFBean {
```

```

@EJB
private BookEJBLocal bookEJBBean;
@EJB
private CategoryEJBLocal catEJBBean;
private String catId;

public StockJSFBean() {
}

public List<Categories> getCategories() {
    try {
        return catEJBBean.getCategories();
    } catch (Exception ex) {
        return null;
    }
}

public String getCatId() {
    return catId;
}

public void setCatId(String catId) {
    this.catId = catId;
}

public List<Books> getBooks() {
    try {
        FacesContext context=FacesContext.getCurrentInstance();
        Map params=context.getExternalContext().getRequestParameterMap();
        this.catId=(String)params.get("categoryId");
        if (this.catId==null||this.catId.equals("")) {
            return bookEJBBean.getBooks(null);
        } else {
            return bookEJBBean.getBooks(new Integer(this.catId));
        }
    } catch (Exception ex) {
        return null;
    }
}
}

```

A StockJSFBean bean getCategories() metódusa a CategoriesEJBLocal interfész getCategories() metódusát hívja meg, mellyel az adatbázisban lévő kategóriák listáját adja vissza, míg a getBooks() metódus a BooksEJBLocal interfész getBooks() metódusát hívja meg, mellyel a paraméterként megadott kategóriába tartozó könyvek listáját adjuk vissza.

Végezetül még néhány képernyőkép a példaalkalmazásból:

The screenshot shows a web browser window titled "Regisztráció - Freedom". The address bar displays "http://localhost:8080/ComputerBookShop". The page has a blue header with the title "Computer Book Shop" and links for "Támogatott nyelvek: [English](#) [Magyar](#)". Below the header is a section titled "Regisztráció". The main content area has a light blue background and contains the text "Kérem, töltsé ki a regisztrációs adatokat:". The registration form includes the following fields and labels:

- Felhasználónév :
- Jelszó : *Jelszó: A mező kötelező.*
- Jelszó újra : *Jelszó újra: A mező kötelező.*
- Keresztnév :
- Vezetéknév :
- E-mail : *E-mail: A mező kötelező.*

Below the form is a "Regisztrálok" button. At the bottom of the page, there is a footer that says "Készült 2010-ben." and a status bar showing "Helyi intranet" and "100%".

13. ábra A registration.jsp regisztrációs oldal, adathitelesítési hibaüzenetekkel

Index - Freedom
http://localhost:8080/ComputerBookShop-v1ar/faces/customer/index.jsp
Live Search
Lap
Eszközök
Index

Computer Book Shop

Támogatott nyelvek: [English](#) [Magyar](#)

Index

Felhasználónév: customer1

Menü [Index](#) [Kosár](#) [Szállítási cím](#)

Kategóriák

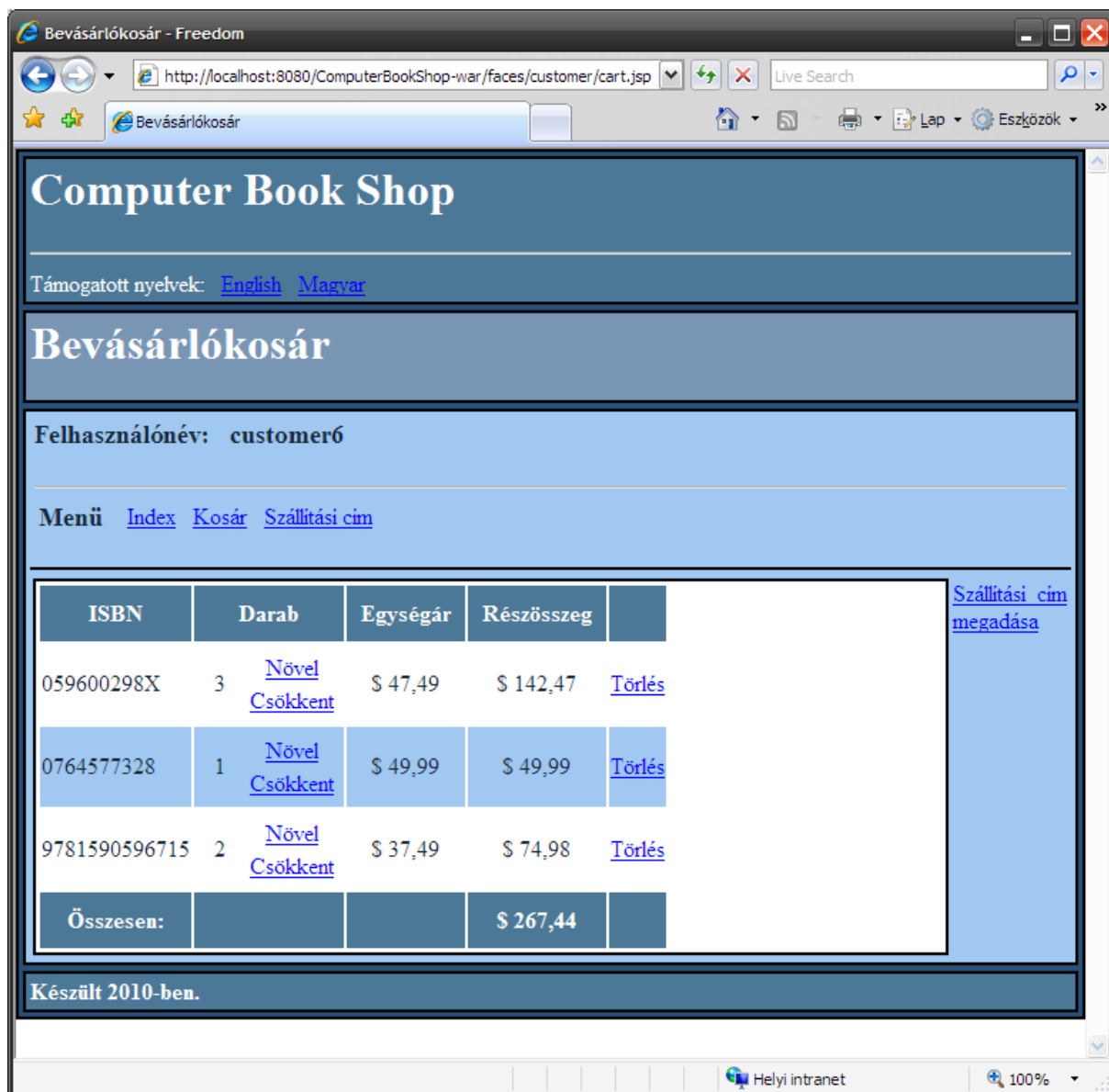
[Minden könyv](#)
[Java](#)
[C/C++](#)
[Delphi](#)
[Databases](#)
[XML](#)

Könyvek

ISBN	Cím	Kategória	Kiadás éve	Oldal	Népszerűség	Mennyiség	Ár
9780470114872	Beginning XML	XML	2007	1255	5	5	\$ 64,99 Kosárhoz ad
059600298X	C++ in a Nutshell	C/C++	2003	808	5	5	\$ 47,49 Kosárhoz ad
0672323087	Object-Oriented Programming in C++	C/C++	2002	1038	3	7	\$ 49,99 Kosárhoz ad
9781590596715	Beginning EJB™ 3 Application Development: From Novice to Professional	Java	2006	510	3	7	\$ 37,49 Kosárhoz ad
0072132930	C++: The Complete Reference	C/C++	1998	1041	2	8	\$ 49,99 Kosárhoz ad
9781847195609	EJB 3 Developer Guide	Java	2008	276	2	8	\$ 34,99 Kosárhoz ad
9781430209638	Beginning Database-Driven Application Development in Java™ EE: Using GlasFish™	Java	2008	418	2	8	\$ 45,99 Kosárhoz ad
0596002920	XML in a Nutshell	XML	2002	634	1	9	\$ 42,49 Kosárhoz ad
0764577328	Beginning SQL	Databases	2005	522	1	9	\$ 49,99 Kosárhoz ad
078214201X	Mastering Delphi 7	Delphi	2003	1011	1	9	\$ 42,99 Kosárhoz ad
ISBN	Cím	Kategória	Kiadás éve	Oldal	Népszerűség	Mennyiség	Ár

Készült 2010-ben.
Kész
Helyi intranet
100%

14. ábra Az index.jsp főoldal, mely alapértelmezésként a tíz legnépszerűbb könyvet jeleníti meg



15. ábra A cart.jsp bevásárlókosár oldal néhány elemmel a kosárban



16. ábra Az order.jsp rendelés adatai oldal, ahol megtekinthető és elküldhető a rendelés

Összefoglalás

Annak ellenére, hogy a Java EE 5 platform nem az egyedüli eszköz vállalati alkalmazások készítéséhez, kétségtelenül az egyik legjobb választás. Olyan alapvető szolgáltatásokat nyújt az alkalmazásfejlesztők számára, mellyel sokkal kényelmesebbé és hatékonyabbá tehető az ilyen jellegű alkalmazások készítése. Bár ezen alkalmazások igen változatosak lehetnek, a Java EE 5 szinte minden üzleti alkalmazástípushoz tud valamilyen támogatást adni.

Egyrészt az EJB és JPA technológiák nagyszerűen használhatók például az adatbázisvezérelt, vagy a változatos kliensekkel rendelkező alkalmazások készítéséhez. Ezek az alkalmazások olyan követelményeket követelnek meg, mint a tranzakció kezelés, biztonság, konzisztens adatkezelés vagy a hálózati kommunikáció.

Másrészt a Webes alkalmazások készítését támogató Servlet, JSP és JSF technológiák is rengeteg hasznos tulajdonsággal rendelkeznek, melyekkel egyszerűen, gyorsan és a Web fejlesztők által megszokott formában készíthetők el az ilyen alkalmazások.

Ezenkívül a nemrég megjelent Java EE 6 platformmal még inkább arra mentek el a készítők, hogy a platform még gazdagabb, de ezzel együtt még hordozhatóbb, még könnyebb súlyú és még egyszerűbben használható legyen. Ezzel együtt nagyobb támogatást nyújt az olyan előretörő technológiáknak, mint a Webes szolgáltatások.

A Java népszerűségéből fakadóan rengeteg szakirodalom és dokumentáció áll az érdeklődő rendelkezésére, így ha valamilyen problémája akad a tanulás vagy a fejlesztés során, szinte biztosan talál segítséget az adott probléma megoldásához.

Remélem sikerült egy rövid, ugyanakkor hasznos és lényegre törő áttekintést adnom a vállalati programozást támogató Java EE 5 platform legfontosabb sajátosságaiból.

Végezetül szeretnék köszönetet mondani témavezetőmnek, Dr. Fazekas Gábornak és szeretteimnek, akik támogatásukkal segítették munkámat.

Irodalomjegyzék

- [1] **Beginning Database-Driven Application Development in Java™ EE Using GlassFish™** by Yuli Vasiliev, Apress, 2008
- [2] **Beginning EJB™ 3 Application Development: From Novice to Professional** by Raghu R. Kodali and Jonathan Wetherbee with Peter Zadrozny, Apress, 2006
- [3] **Beginning Java EE 5 From Novice to Professional** by Kevin Mukhar and Chris Zelenak with James L. Weaver and Jim Crume, Apress, 2006
- [4] **Beginning JSP 2: From Novice to Professional** by Peter den Haan, Lance Lavandowska, Sathya Narayana Panduranga and Krishnaraj Perrumal, Apress, 2004
- [5] **Core JavaServer™ Faces, Second Edition** by David Geary, Cay Horstmann, Prentice Hall, 2007
- [6] **EJB 3 Developer Guide** by Michael Sikora, Pact Publishing, 2008
- [7] **EJB 3 in Action** by Debu Panda, Reza Rahman, Derek Lane, Manning, 2007
- [8] **Java EE 5 Development using GlassFish Application Server** by David R. Heffelfinger, Pact Publishing, 2007
- [9] **JavaServer Faces in Action** by Kito D. Mann, Manning, 2005
- [10] **Pro EJB 3: Java Persistence API** by Mike Keith and Merrick Schincariol, Apress, 2006
- [11] **The Java EE 5 Tutorial**, Sun Microsystems, 2007

Függelék

A. Az adatbázist létrehozó SQL utasítások

```
create table users (  
    username varchar(32) primary key,  
    password char(32) not null,  
    firstname varchar(40) not null,  
    lastname varchar(40) not null,  
    email varchar(80) not null  
);  
  
create table groups (  
    group_id integer primary key,  
    group_name varchar(30) not null  
);  
  
create table usergroups (  
    username varchar(32),  
    group_id integer,  
    primary key (username, group_id),  
    foreign key (username) references users(username),  
    foreign key (group_id) references groups(group_id)  
);  
  
create table orders (  
    order_id integer primary key,  
    username varchar(32) not null,  
    order_price real not null,  
    order_date date not null,  
    zipcode varchar(20) not null,  
    city varchar(40) not null,  
    street varchar(40) not null,  
    addr_number varchar(20) not null,  
    foreign key (username) references users(username)  
);  
  
create table categories (  
    category_id integer primary key,  
    category_name varchar(30) not null  
);  
  
create table books (  
    isbn varchar(24) primary key,  
    title varchar(80) not null,  
    category_id integer not null,  
    date_of_issue integer not null,  
    pages integer not null,  
    popularity integer not null,  
    quantity integer not null,  
    price real not null,  
    foreign key (category_id) references categories(category_id)  
);  
  
create table items (  
    order_id integer,  
    isbn varchar(24),  
    units integer not null,  
    unit_price real not null,  
    primary key (order_id, isbn),  
    foreign key (order_id) references orders(order_id),  
    foreign key (isbn) references books(isbn)  
);  
  
create table shoppingcarts (  
    username varchar(32),  
    isbn varchar(24),  
    units integer not null,
```

```

        unit_price real not null,
        primary key (username, isbn),
        foreign key (isbn) references books(isbn)
);

create table pk_table(
    pk_name varchar(30) primary key,
    pk_value integer
);

create view v_usergroups as
    select users.username, users.password, groups.group_name
        from users, groups, usergroups
        where users.username=usergroups.username and
              groups.group_id=usergroups.group_id;

```