**EJB (Enterprise Java Beans):** Fontos része a J2EE –nek. A J2EE komponens alapú architektúrával biztosít több felhasználós eszközöket az alkalmazáslogika fejlesztéséhez.

Az EJB három kategóriába sorolható: Mindegyik típusnak megvan a maga sajátossága a tervezés szempontjából (cél, életciklus, scope, stb).

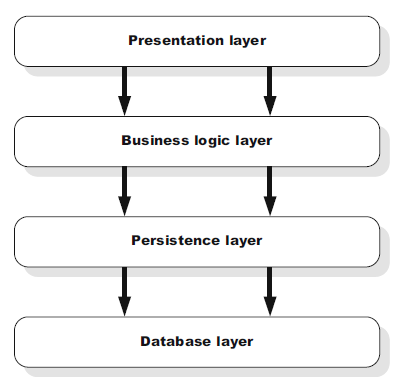
* Session Bean: Ez lehet stateful vagy stateless. Ez kevésbé erőforrás pazarló, mint az Entity Bean, mivel ez rögtön elpusztul, mihelyst a felhasználói session véget ér.
* Entity Bean: A perzisztens adatok tárolása a feladata. Felhasználói adatokat kimenthetünk adatbázisba Entity Beaneken keresztül, és később visszatölthetjük a beanbe az adatokat az adatbázisból.
* Message Driven Bean: JMS (Java Messaging Service) környezetben használatosak.

**EJB előnyei:**

* Gyors és egyszerű fejlesztést tesz lehetővé.
* Az Alkalmazás Szerver/EJB konténer sok rendszer szintű szolgáltatást biztosít, mint például tranzakció kezelés, loggolás, perzisztencia támogatás, kivétel kezelés. A fejlesztőnek csak az üzleti logika implementálásával kell foglalkoznia.
* Az EJB konténer felügyeli az EJB példányok életciklusát, tehát a fejlesztőknek nem kell a létrehozásukkal és törlésükkel foglalkozni.
* Az EJB3 az egy POJO, némi extra funkcióval. Ezen funkciók mindaddig fedve vannak, amíg nincs rájuk szükség.

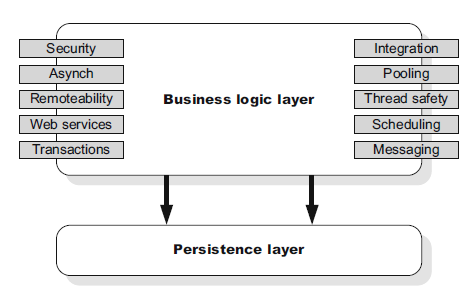
**Rétegezett architektúra és EJB:** A szerveroldali alkalmazásoknál a legdominánsabb tervezési minta a rétegezett tervezés. Egy rétegezett architektúrában a komponenseket sorokba (tier) soroljuk. Minde sornak (Tier) megvan a jól definiált feladata. Minden réteg kiadja a feladatot az alatta lévő rétegnek.

**Tradicionális 4-tier szerver architektúra:**



Ahogy a fenti ábrán is láthatjuk, 4 rétegből épül fel.

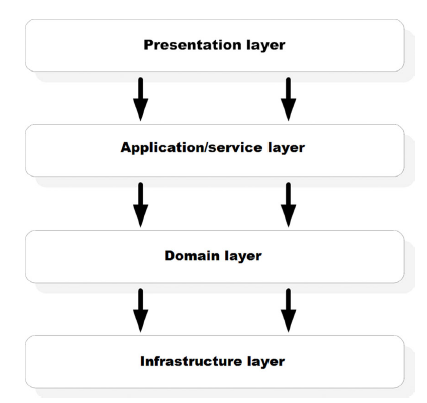
* Megjelenítő (Presentation) réteg: Ez felelős a grafikus megjelenítésért, és a felhasználói input adatok kezeléséért. Ez a réteg minden kérést tovább ad az alatta lévő Üzleti logika (Business Logic) rétegnek. Ez lehet böngészős vagy desktop alkalmazás is.
* Üzleti logika (Business Logic) réteg: Az alkalmazás szíve. Itt történik keresés, listázás, maga az üzleti logika implementációja.
* Perzisztens (Persistence) réteg: Az Üzleti logika réteg ezen rétegen keresztül hoz fel, és ment ki adatokat az adatbázisból. Ez a réteg magas szintű (OO) adatbázis kezelést biztosít az adatbázis réteg felé.
* Adatbázis (Database) réteg: Itt történik az adatbázis kapcsolódás, és alacsony szintű kezelés.



Látható, hogy az Üzleti logika rétegbe nagyon sok független szolgáltatás található, ezeket kedvünkre használhatjuk.

Fontos megemlíteni, hogy egy EJB az nem perzisztens és adatbázis réteg technológia. Ahogy a fenti ábrán is látható, az EJB bizonyos szolgáltatások biztosításával segíti a fejlesztést, ezek az üzleti logika fejlesztésénél fontosak.

**DDD (Domain-Driven Desing) arcitektúra:**



Manapság inkább ezt az architektúrát használják. A Megjelenítési réteg felelős a megjelenítésért, és az Alkalmazás/szolgáltatás réteg felé továbbításért. Az Alkalmazás/szolgáltatás réteg általában elég vékony, főként arra szolgál, hogy kommunikációs csatornaként szolgáljon a Megjelenítési réteg és a Domain réteg között. A Domain rétegben van az alkalmazás adatmodellje, tartalmazva entitásokat, gyártókat, és repozitorikat. Az Infrastruktúra réteg kezeli az adatbázis és más perzisztenciát használó technológiákat. A domain objektumok tartalmazzák az üzleti logikát, ezek az objektumok egy JPA Entitásként (@Entity) feleltethetőek meg.

**Teljesítmény és skálázhatóság:** Komoly EJB konténer skálázhatósági és hibatűrési szolgáltatásokat is nyújt cluster-es környezetben. Az EJB alkalmazás modell egyik legfontosabb képessége pont ezeknek a szolgáltatásoknak az igénybevételében van, egy EJB modellben írt alkalmazás a megfelelő alkalmazásszerveren telepítve skálázható (tehát kapacitása további számítógépek hozzáadásával elvileg korlát nélkül növelhető) és hibatűrő lesz (tehát egy gép kiesése a cluster-ből csak kapacitáscsökkenést okoz, de a rendszer működik tovább).

Az EJB alkalmazásmodellnek azon ígérete, hogy alkalmazásaink extrém skálázhatósági követelményeknek is eleget tudnak majd tenni (megfelelő számítógéprendszeren és alkalmazásszerverrel) komoly megkötésekkel jár. Ehhez az kell, hogy az alkalmazás ne tartalmazzon semmit, ami egy bizonyos számítógéphez köti és az alkalmazásszerver nem tud róla. Példák a megkötésekre:

Az EJB nem hozhat létre végrehajtási szálakat, hiszen egy végrehajtási szálat nem lehet áttenni egy másik számítógépre.

Az EJB nem használhat fájlokat a helyi fájlrendszeren, hiszen ha a bean következő hívásakor egy másik számítógépen fut le, a fájl nem lesz elérhető. Fájlszerver is óvatossággal használandó, két EJB hívás között a fájl nem hagyható nyitva, mert a helyi erőforrás (fájldeszkriptor) állapotáról a szervercsoport másik tagja nem tud, márpedig túlterhelés vagy hiba esetén a bean egy másik gépre kerülhet.

Az EJB nem kezelhet hálózatot se, hiszen pl. egy socket olyan erőforrás, amit nem lehet más gépre mozgatni.

**EJB típusok:** Ebben a világban egy komponensnek egy Bean-t tekintünk. Két típusú Beant különböztetünk meg, a Session Bean, és a Message-driven Beanek formájában. A típusok elkülönítésének az a célja, hogy biztosítsa, hogy ne terheljük túl őket szolgáltatásokkal.

**Session Bean:** Ezeket a kliensek használják, hogy segítségükkel végrehajtsanak üzleti tevékenységeket, mint például leellenőrizni a vásárlók vásárlásait. (imply=maga után von). Egy szerver összeomlást nem él túl. A Session Beaneknek 3 fajtájuk van:

* Stateless: Ez a bean lehetővé teszi metódusok kontextus nélküli hívását, tehát a metódushívások között a konténer nem őriz állapotot, a kontextus csupán a hívás során létezik. A bean egyedváltozói tehát nem használhatók hívások közötti állapot tárolására. A bean állapot nélküli volta lehetővé teszi az EJB konténernek, hogy a bean objektumegyedet kedve szerint hozza létre és pusztítsa el a kliensprogram tudomása nélkül. Például terhelésmegosztási mechanizmust használhat és két hívás között úgy dönthet, hogy a következő hívást a szervercsoport másik gépe szolgálja ki. A kliens következő hívását az első szerver EJB konténere átirányítja a másik szerver EJB konténeréhez, amely elővesz egy egyedet ebből a típusú állapotnélküli kapcsolat bean-ből és meghívja rajta a bean metódust. Ebben az extrém példában a bean egyed még csak nem is ugyanazon a számítógépen volt, ez is mutatja, miért nem lehet az egyedváltozókat állapotmegőrzésre használni.
* Stateful: Automatikusan elmenti a Bean állapotát. A Bean megőrzi a klienshez rendeltségét és egyedváltozóinak állapotát két hívás között.

remote.myMethod();

...

remote.myMethod();

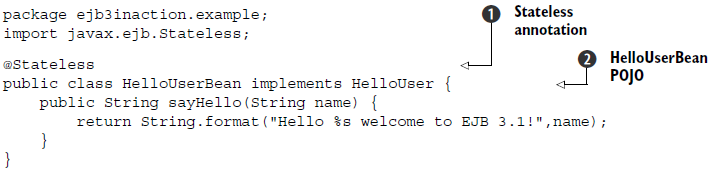
az állapot nélküli bean-nél a két hívás két külön bean-hez futhat be, amelyeknek állapotai (egyedváltozói) különbözőek lehetnek. Ha véletlenül a két hívás ugyanahhoz a bean egyedhez futna be, akkor se garantálja semmi, hogy a két hívás között egy másik kliens nem használta a bean-t. Állapottal rendelkező bean esetén mindkét hívás azonos egyedváltozókat talál (vagy legalábbis konzisztens egyedváltozókat, hiszen a hívások meg is változtathatják azokat) és más kliens véletlenül nem használhatja ezt a bean egyedet, csak ha bean létrehozója átadta a bean referenciáját valaki másnak (bean referenciák átadásáról később). Természetesen a bean egyed nem feltétlenül ugyanaz, hiszen a bean-t esetleg át kellett mozgatni egy másik számítógépről vagy helyre kellett állítani tárolt formából. A SessionBean interfész végül is a Serializable leszármazottja, tehát a bean mozgatható az alkalmazásszerverek között vagy tárolóba menthető és onnan visszaállítható.

* Singleton: Állapotok mentésére használható, minden kliens osztozkodik rajta, és az alkalmazás életét végigkíséri, végig elérhető.

(invoke-felhasznál) A Session Beanek használhatóak két formában, vagy lokális vagy távoli(remote) elérésűként. A Stateless vagy a Singleton Session Beanek használhatóak REST vagy SOAP szolgáltatásként is.

**Message-driven Bean:** A Session Beanekhez hasonlóan ezekkel is üzleti logikát lehet megvalósítani. Egy bizonyos dologban nagyon eltérőek viszont: Ennek a metódusait a kliens nem hívhatja, nem érheti el közvetlenül. Ezt a bean típust elsősorban Java Messaging Service (JMS) üzenetek kezelésére tervezték. A JMS-ről most elég annyit tudnunk, hogy úgy működik, mint egy levelezőlista: a JMS-ben topic-nak hívott témákra akármilyen adatstruktúrát lehet küldeni, amelyek sorba állnak, majd a témára feliratkozott alkalmazások megkapják őket. Eseményvezérelt bean lehet feliratkozva egy témára, ekkor a JMS üzenetek hatására aktivizálódik. A bean életciklusát a konténer és a beérkező üzenetek vezérlik. Üzenet érkezésekor a konténer szerez egy szabad bean-t (gyárt egyet vagy elővesz egyet az egyedtárolóból) és odaadja az üzenetet. Minthogy a bean-t kliensek nem hívhatják meg, se vezérlő, se távoli interfésze nincsen, csak a bean osztály.

**Hello User példaalkalmazás:**



(invocation-megszólítás, segítségül hívás)

Az összes publikus metódus meghívható a beanre. Ezzel óvatosan kell bánni, mert mondjuk a cégÖsszesTitkosInformációja() metódus érdemes ha private. A fent látható osztály az egy POJO. A @Stateless annotáció az egy metaadat, ennek segítségével átalakítják a POJO-t egy Stateless EJB-re.

**Annotációk előnye:** Annotációkkal metaadatokat rendelhetünk az osztályokhoz, adattagokhoz és metódusokhoz. Amikor a külső feldolgozó(jelen esetben az EJB konténer) értelmezi ezeket, akkor ő kiegészíti a kódot. Lényegében a programozó specifikálja minek kellene történnie, a rendszer pedig hozzá adja a kódot a háttérben. Ezt hívják dekleratív stílusú programozásnak.

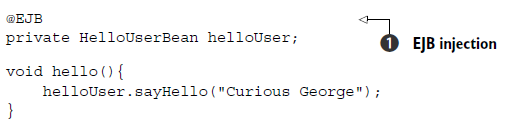


Az annotációval egy egyszerű POJO-ból EJB készíthető.

(approach-megközelítés, közeledés)

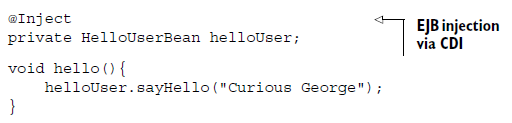
**Intelligens Defaultok:** EJB egy kicsit másképp kezeli az alapértelmezett viselkedéseket, mint más keretrendszerek. Például Springben akkor kapsz valamit, ha kérsz. Az EJB3 megpróbálja csökkenteni a konfiguráció mértékét azáltal, hogy Default beállításokat alkalmaz, ott ahol csak lehet. Például a „Hello World” komponens alapértelmezetten szálbiztos, pool-olva van, és tranzakció menedzselt, ezeket mind anélkül teszi, hogy beállítottuk volna. Továbbá ha aszinkronitást szeretnénk, vagy web service-t beállítani, akkor annotációkkal könnyedén megoldható.

**Dependency Injection vs JNDI:** EJB3 –ban már van Dependency Injection, ami azt jelenti, hogy egy EJB injektálható egy másik Java EE komponensbe, illetve egy Java EE komponens is beinjektálható egy EJB-be.



Ha a HelloUserBeanünket szeretnénk beinjektálni egy másik EJB, Servlet, vagy más Java EE komponensbe, akkor az @EJB annotációval ez megtehető. Az annotált változóba beleinjektálja a Beant. A háttérben JNDI-t használ.

**CDI vs EJB injection:**

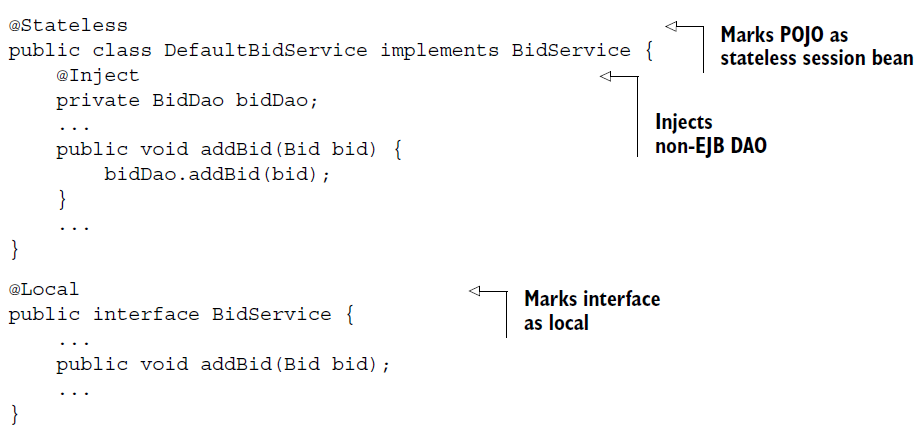


A fenti képen CDI injektálás látható. Ez annyival másabb az EJB injektálástól, hogy ezzel több mindent lehet beinjektálni, EJB picit korlátozottabb. Úgy tűnhet, hogy ezt az injektálást használhatjuk minden Java EE komponensre, viszont ennek vannak korlátai. Sajnos ez nem működik remote EJB-kre. Ellenben az @EJB felismeri, hogy egy Bean mikor local és mikor remote, és visszatér azzal a típussal, ami kell. Viszont @Injection használatos akkor, amikor csak lehetséges.

(retrieve-elhoz, helyrehoz)

**Tesztelhető POJO komponensek:** Mivel minden EJB lényegében egy szimpla POJO, ezért könnyen unit-tesztelhetőek. Használható CDI, hogy egy EJB-t beinjektáljunk egy teszt osztályba.

**Egyszerű példa:**



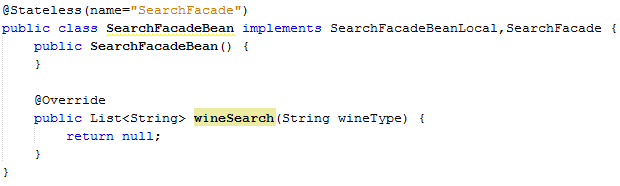
@Stateless jelentése: A konténer automatikusan szálbiztonságot, tranzakciókezelést és poolingot biztosít. A szélbiztonság és tranzakció kezelés azt jelenti, hogy bármely erőforráshoz hozzáférhet, a tranzakció kezelés és szálkezelés megvan oldva.

@Local jelentése: Lokálisan hozzáférhetünk a beanhez az interfészen keresztül. Az EJB komponensek általában egy közös alkalmazásban szerepelnek, ezért tökéletes megoldás most a local. Alternatív megoldás a @Remote és a @WebService. A remote ejb az más JVM-ből való elérést takar. A @WebService annotációval ha nem-java kliensekről van szó(SOAP,php, .net).

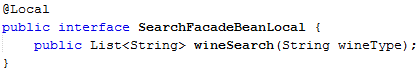
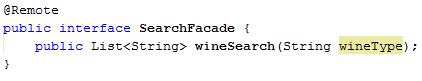
@Inject: A Bid szolgáltatásnak a perzisztáláshoz szüksége van a Bid DAO-ra, amelyet injektálással kap meg. A Bid DAO az nem EJB komponens. Tehát ezzel beinjektáljuk A BidServicebe a BidDAO-t.

**Session Bean:** Már említésre került, hogy 3 típusa van.

**Stateless Session Bean:** Először is egy szimpla Java osztályt kell létrehoznunk.

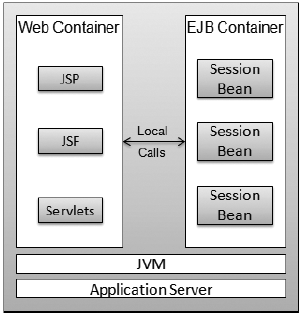


A fenti osztályból az annotáció segítségével Stateless Session Beant készítettem. Ennek a Beannek az lesz a szerepe, hogy a klienseknek különböző keresési szolgáltatásokat nyújtson, amelyekkel elérhető borokat kérhet le. A felhasználók beírják vagy kiválasztják a bor kritériumokat, majd átadódik a Beannek. A SearchFacadeBean a szolgáltatás futtatásakor segítséget kér az adatbázistól, és úgy szolgáltat vissza adatokat.

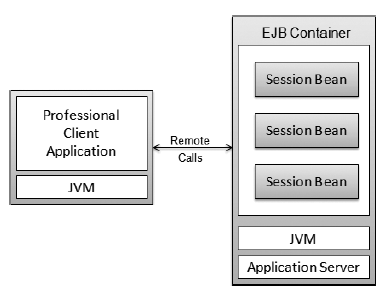


Két féle interfész adható meg egy Beannek.

Ha az interfész nincs ellátva annotációval, akkor automatikusan minden publikus metódus local lesz. Ha a webes alkalmazásoknak különböző JVM-ben kell futniuk, akkor kell, hogy @Remote legyen, ha ugyan azon a JVM-en fut, akkor @Local, ami az automatikus, ha nincs interfész a Beanhez.



A fenti ábrán a @Local –ra láthatunk példát, amiben egy JVM-ben fut minden.



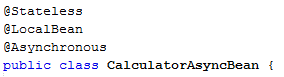
A fenti ábrán pedig az látható, amikor két külön JVM-ből hívjuk a Beaneket.

Aszinkron metódusok használata: Az aszinkron metódus rögtön visszatér a hívóhoz, anélkül, hogy ki kellene várni, hogy a metódus végrehajtja a feladatát. Általában processzor intenzív, háttér taskok esetén használatos, mint nyomtatás, vagy e-mail küldés. Amikor egy Session Bean végrehajt egy aszinkron metódust, a konténer rögtön átadja a vezérlést a hívónak.

Hogyan készíthetünk aszinkron metódusokat?

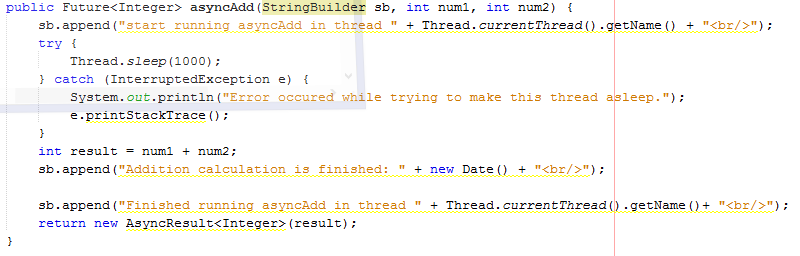
Ha az osztályt ellátjuk @Asynchronous annotációval, akkor az összes metódus aszinkron lesz. Egy aszinkron metódus visszatérési értéke csak void vagy java.lang.concurrent.Future<V> interfész implementáció lehet. A void visszatérésű aszinkron metódusok nem dobhatnak application exception-t, viszont a másik esetben dobhatunk.

Az alábbi példában létrehozok egy aszinkron java beant, amely két metódust tartalmaz, az egyikkel két számot lehet összeadni, a másikkal pedig kivonni. A lényeg abban van, hogy a háttérben még altatjuk is a szálat, ezzel szimulálva, mintha sokáig dolgozna a háttérben a megoldáson.

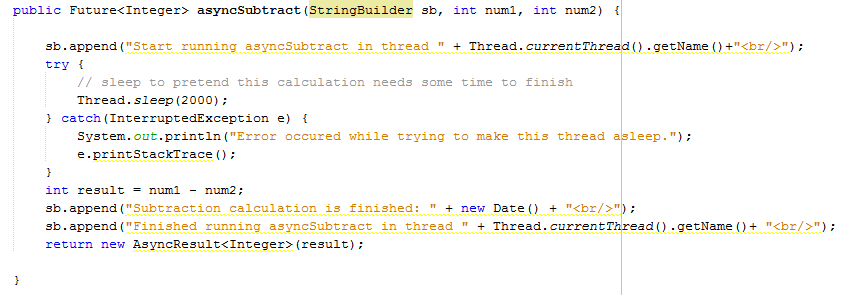


A fenti képen az osztály definiálását láthatjuk, mivel itt adjuk meg az aszinkron annotációt, ezért minden metódus aszinkron lesz, amit ebben az osztályban definiálunk.

Az alábbi két képen a két aszinkron metódus definícióját láthatjuk.



A fenti képen a paraméterben kapott számokat adom össze, persze előtte várok egy másodpercet. Ami még észrevehető, ha egy aszinkron metódusnak van visszatérési értéke, akkor azt az értéket az AsyncResult<> csomagoló osztályban kell visszaadni.

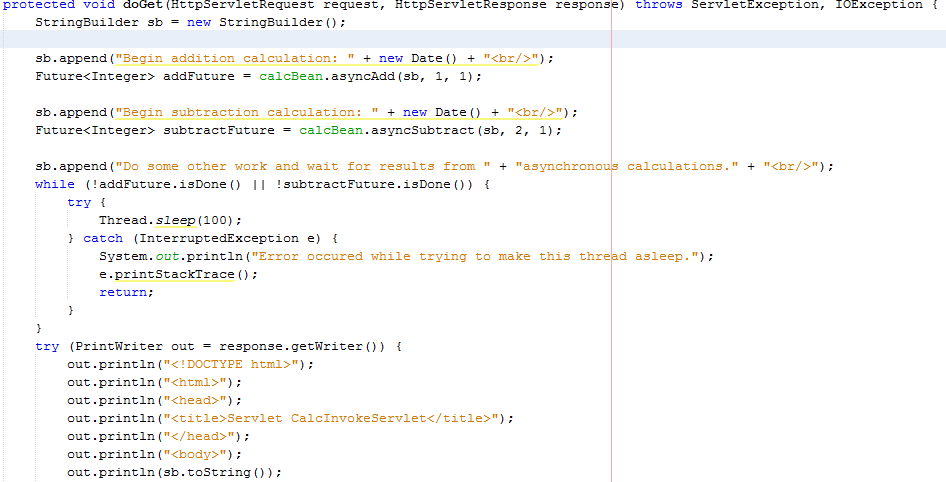


Ezek után csináljunk egy klienst, ami meghívja ezeket, a kliensünk pedig egy Servlet lesz.

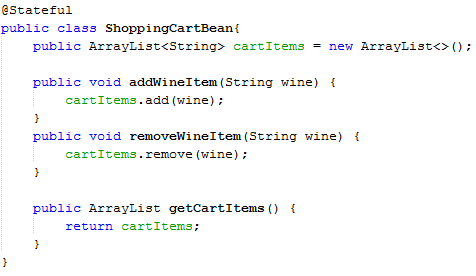


A Servlet osztályunkba beinjektáljuk a Stateless Beant, aminek meghívhatjuk az aszinkron metódusait.

Az alábbi képen a get kéréseket kiszolgáló metódus kódját láthatjuk:



A kliens, amikor ráhív egy aszinkron metódusra, aminek van visszatérési értéke (Future<>), az rögtön visszatér hozzá egy Future<Value> példánnyal, így a vezérlést is visszakapja. De mi van benne a visszakapott objektumban, ha a háttérben az aszinkron metódus még dolgozik a visszatérési értéken?



A megkapott objektumnak hívhatjuk különböző metódusait:

isDone(): Ez a metódus igaz értékkel tér vissza, ha az aszinkron metódus már végzett a futásával

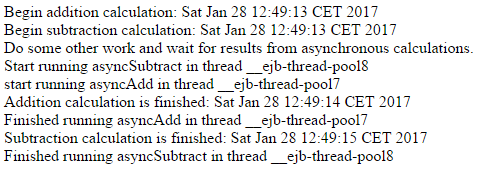
cancel(boolean): Ez nem lövi ki a szálat, csak a cancelled flag értékét állítja be, amely lekérhető az isCancelled() metódussal.

get(): Az isDone() true visszatérése után szokták meghívni, ebből nyerhető ki az eredmény.



A fenti példában a számítás eredményét nem vettük ki, viszont a fenti képen látható módon ezt megtehetnénk.

Tehát a fenti Servlet eredménye a következő:



Az eredmény levezetése: Az első két sort a Servlet adja hozzá a StringBuilderhez, mielőtt meghívja az adott metódust. A két metódus után látható, hogy a vezérlés rögtön visszaadódik a kliensnek (ami jelen esetben a szervlet), és kiírja, hogy csinál valamit és vár. Egészen addig vár a while ciklussal, amíg mindkét metódus a háttérben nem végez a teendőivel, majd azok is hozzáírnak futásuk közben a StringBuilderhez, és ezt a StringBuildert a tartalmával adjuk át a kliensnek megjelenítésre.

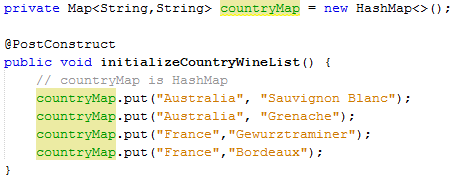
Ezek után, amit tárgyalni kell a Stateless Session Beanekkel kapcsolatban, az az Életciklus Callback függvények: Léteznek események, például a Bean megszületése és elpusztulása, amelyekre érdemes lehet eseményt írni.

Például a SearchFacade Beannek szüksége lehet adatbázis kapcsolat kialakítására és lekérdezésre, amikor létrejön, illetve elpusztulásakor annak az adatbázis kapcsolatnak a lezárására.

Az EJB konténer fogja meghívni ezeket a metódusokat, akkor amikor a Bean életciklusa megfelelő állapotban van(születés, elpusztulás).

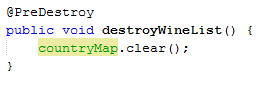
Stateless Session Beanekhez két féle callback létezik:

* PostConstruct:



A Bean születésekor hívja meg az EJB konténer (amikor az EJB konténerben létrejön belőle egy példány). Ha a Bean igényel injektálást, akkor azokat fogja először elvégezni a konténer, aztán ezt a metódust, majd ezeket követően lehet csak ráhívni a Bean üzleti logikát megvalósító metódusaira.

* PreDestroy:



Ez pedig a Bean elpusztulásakor hívódik meg (a konténer a példány kisöprése előtt hívja meg, mielőtt kisöpörné az objektum poolból). Arra szokás használni, hogy lezárjunk olyan connection poolt, amit dependency injectionnel szereztünk, illetve más erőforrásokat is felszabadíthatunk. Igazából a fenti példának, ahol a countryMap-et ürítjük ki semmi haszna, hisz a Bean elpusztulásakor ez is felszabadulna, hisz nincs rá más referencia, de most a példa kedvéért ezt csináltam.

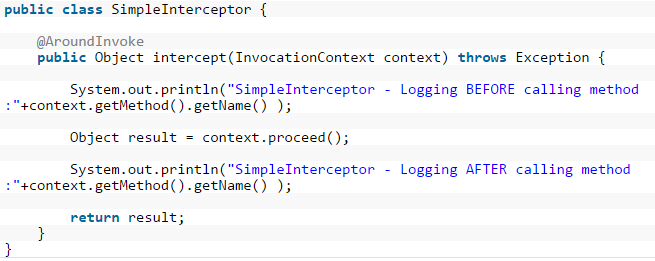
A következő, amit át kell néznünk, az az Interceptor (gyűjtő, megszakító) használata Stateless Session Beanek esetén: (interpose->közbelép)

Ennek szerepe, hogy az üzleti logikát megvalósító metódusok meghívása előtt vagy után becsomagolhassunk saját metódusunkat. Használata:

Nagy összegű utalás esetén ellenőrzést végezhetünk.

Loggolás a metódus előtt és után.

Egy egyszerű interceptor készítése:

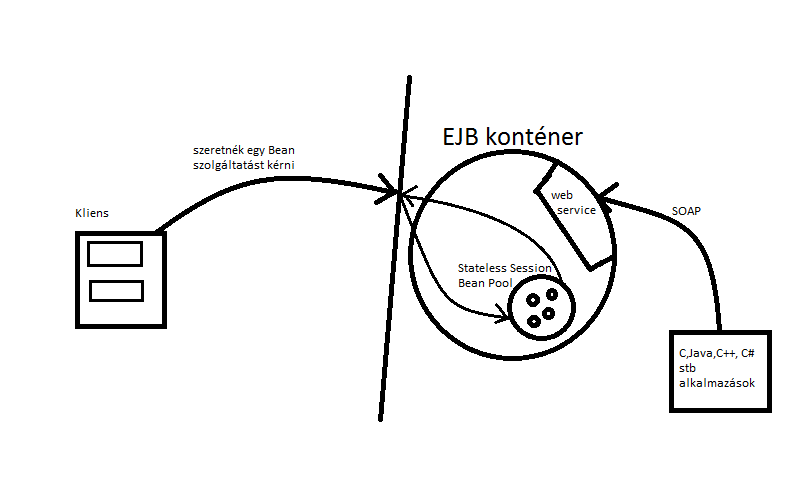


Készítettünk egy osztályt, amiben van egy olyan metódus, ami interceptorként fog meghívódni. Paraméterben kapott InvocationContext objektumot sok mindenre használhatjuk, de azokra majd később térek ki. Itt most arra használjuk, hogy tovább adjuk a vezérlést, a többi interceptornak, vagy a bean metódusnak. Amit a context.proceed() metódushívás előtt definiélok, az a Bean metódushívás előtt fog meghívódni, amit utána, az pedig a Bean metódus lefutása után. Egy interceptor osztálynak, mint a fenti képen látható akármennyi metódusa lehet, viszont ezek közül csak egy lehet interceptornak felannotálva a @AroundInvoke annotációval. A fenti interceptor úgy van megírva, ha nincs több interceptor, és lefutott a bean metódus, akkor annak a visszatérési értékével tér vissza ez az interceptor osztály.(Object result = context.proceed()).

Mi nem tudjuk az EJB metódus visszatérési értékét, ezért a proceed() Objecttel tér vissza. Miért jó az, hogy az interceptor visszatér a Bean metódushívás eredményével? Azért mert így ezt tovább tudja adni a interception híváslánc következő tagjának vagy a kliensnek.

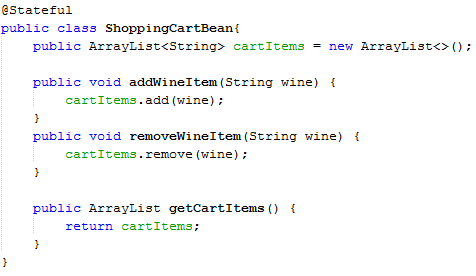
https://www.javacodegeeks.com/2013/07/java-ee-ejb-interceptors-tutorial-and-example.html

**Beanek működési ábrája:**



Ami látható a fenti ábrán, hogy a kliens(ami sokszor egy szervlet, szolgáltatásokat vesz igénybe a beaneken keresztül, a beanek poolokban vannak, így van egy példányosított készlet belőlük, amit a konténer szabályoz. Stateless Session Beanek esetén nem biztos, hogy két különböző kérést ugyaz a a Bean példány fog kiszolgálni. Látható, hogy a webszolgáltatások is a beanekre épülnek. Tehát itt nem html válasz várható, hanem főként XML SOAP, és ezen a felületen is igénybe lehet venni a szolgáltatást).

**Stateful Session Beanek:** Főként egy webshop kosár az ami jó szemléltető eszköz lehet egy Stateful Session Beanre. A konténer a Stateful session Beaneket is Poolban tárolja, viszont itt a SessionID-k alapján minden Sessionhoz létezik egy ilyen objektum. Ez nagyon sok user esetén leterhelheti a memóriát. Megoldás, hogy azon sessionhoz tartozó objektumokat, amely session egy ideje inaktív, azokat perzisztálja a háttérbe, majd ha Marika néni újra géphez ól és folytatja a webes vásárlást, akkor visszatölti a memória poolba. Tehát ez Sessionhöz van kötve, tehát amíg a session él, addig ez az objektum is él.



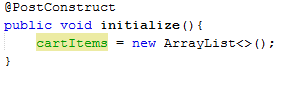
A Stateful Session Beant egy webshop kosáron fogom szemléltetni. Ami látható, hogy a Beannek van egy lista adattagja, amiben a kosárba rakott elemeket tárolja(amíg ugye él a sessione a usernek, mert ha megszűnik, akkor a Stateful Session Bean is felszabadul a memóriából).



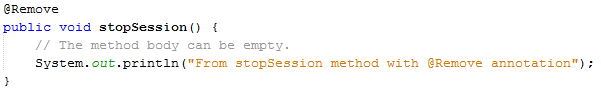
A fenti ábrán a Beant használó kliens szervlet implementációja látható. A szervletbe beinjektáljuk a Stateful Session beant, majd minden kérésre hozzáadunk egy almát a kosárhoz, ezt követően pedig olyan html-t küldünk vissza a kliensnek, amely kilistázza a kosár tartalmát. Itt a servel mindig növeli az almák számát a Beanben, és ugyebár ezt a Beant fogja megkapni Session alapján mindig, nem pedig más példányt.

**Életciklus metódusok Stateful Session Bean esetén:**

* @postConstruct: Ez akkor fog meghívódni, amikor inicializálódik a Bean, tehát ha a Beanbe más Beaneket injektálunk azok még ez előtt fognak lefutni, viszont az üzleti logika metódusai csak ez után hívódhatnak.



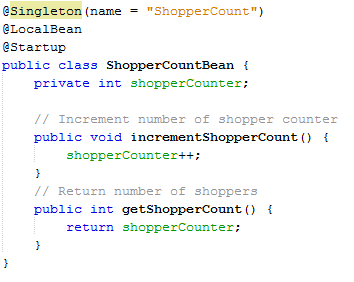
* @PreDestroy: Főként erőforrás felszabadításra és adatok perzisztálására szokták használni, mielőtt felszabadulna a Bean a memóriából.
* @prePassivate: Ezzel az annotációval ellátott metódus az előtt hívódik, mielőtt a konténer passzív állapotba menti a Stateful Session beant (ezt ugye azért csinálja, hogy memóriát spóroljon, és kimenti az objektumot háttértárra). Tehát mielőtt kimentené az objektumot, az előtt közvetlenül hívja meg ezt a metódust.
* @postActivate: Ezzel az annotációval ellátott metódus akkor hívódik meg, amikor egy passzív állapotba mentett Stateful Session Beant újra betölt a memóriába a konténer.
* @Remove: Az a lényeg, hogy az ilyen annotációval ellátott metódus meghívása után, a konténer felszabadítja az objektumot a poolból. Ez a metódus lehet üres is, hisz az annotáció hatására elérjük vele a célunkat úgy is.



**EJB hibakezelés:** Két típust különböztetünk meg:

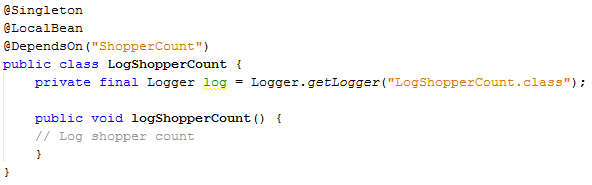
* Application Exception: Üzleti logikában fellépő hibák kezelésére, például nem megfelelő hitelkártya számot adott meg a user. Ezen hibaosztályoknak a java.lang.Exception osztályból kell leszármaznia.
* System Exception: Rendszer szintű hibák sorolhatóak ide, mint például JNDI error. A System Exception osztályoknak a java.rmi.RemoteException vagy a java.lang.RuntimeException leszármazottjának kell lennie. Az alkalmazás szintű exceptionok elkapják a System Exceptionoket.

**Singleton Session Bean:** Ebből egy példány létezik alkalmazásonként. Egy szerver összeomlást ez sem él túl, ez is a memóriában tárolódik.



A fenti Singleton Session Bean arra szolgál, hogy az összes éppen vásárló embert megszámoljuk, egy helyen van ezáltal eltárolva, hogy hányan vásárolnak most a webshopunkban. A @Startup annotáció hatására a konténernek kötelessége inicializálni ezt a Beant még az alkalmazás startup szekvenciájában.

Ha a Singleton Sessiob Beanek hivatkoznak egymásra, például a logolo Beanünk használ egy olyan Beant, ami @Startuppal van ellátva, akkor azt a @DependsOn annotációval jelezni kell mint függőséget.

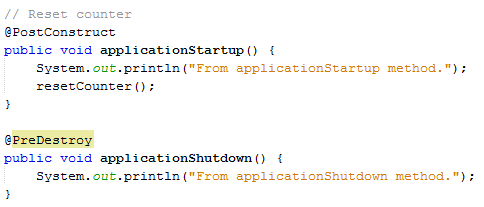


A másik két fajta Session Beannel ellentétben a Singleton Session Beanben nem kell implementálni kötelezően a javax.ejb.SessionSynchronization interfészt, vagy annotálni.

Itt is ugyan úgy létezik hozzá @Local és @Remote interfész.

**Életciklus metódusok Singleton Session Bean esetén:**

* @PostConstruct: Ez akkor fog meghívódni, amikor inicializálódik a Bean, tehát ha a Beanbe más Beaneket injektálunk azok még ez előtt fognak lefutni, viszont az üzleti logika metódusai csak ez után hívódhatnak.
* @PreDestroy: Az alkalmazás leállításakor fog meghívódni. Ez természetesen figyel a felszabadítási sorrendre, tehát a Logger Singleton Bean ami függ a Counter Beantől az fog hamarabb eltávolítódni, mint a Counter.

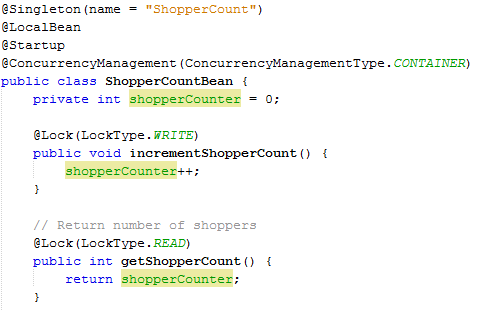


A Stateless Session Beanhez hasonlóan a Singleton Bean sem kerül sosem perzisztens állapotba mentésre, ezzel kisöpörve a memóriából, ez is mindig ott van a memóriában.

**Konkurencia kezelés:** Egy Singleton Session Beanből egyetlen példány él, és valahogy gondoskodni kell a konkurens hozzáférésről.

Konkurencia menedzsment két fajtája:

* Konténer által kezelt konkurencia:



A konténer kezelt konkurenciát úgy állíthatjuk, hogy @Lock annotációt használjuk a metódus felett, és itt állíthatunk írási és olvasási zárolást. A Default beállítás a @Lock(LockType.WRITE). Látható, hogy a változó növelésére írás védettséget tettünk, míg olvasásra olvasás védettséget.

* Bean által kezelt konkurencia: Ilyenkor a bean fejlesztőjének kell gondoskodnia a lockolásról volatile változókkal és synchronize blokkokkal és metódusokkal.

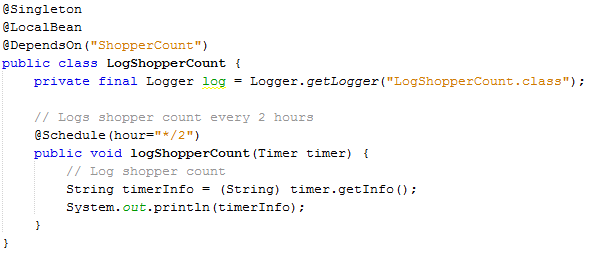
**Idő szolgáltatás (Timer Service):** Az EJB Timer Service az egy konténer által kezelt szolgáltatás. Lehet naptár alapú időzítés, pontos idő, valami után.

Az időzítőket alkalmazás szintű eseményekre használjuk, ne real-time megoldásokra.

Tipikus felhasználási területei:

* Költséges műveletek végrehajtása minden nap 9 órakor (sok nyomtatás)
* Minden január elsején történjen valami

Timer nem lehet Stateful Session Bean!!!!



A fenti ábrán például 2 óránként logoljuk, hogy hányan vásárolnak abban az időpillanatban. Ha a felannotált metódusnak paraméterben beteszünk egy Timer objektumot, akkor azt megkapjuk a konténertől, és információkat kérhetünk le tőle.

Az alábbiakban néhány Calendar alapú idő megadást láthatunk:



**Web Service:** Ahogy egy már korábbi ábrán látható volt, webszolgáltatásokra is kitűnően alkalmazható az EJB konténer, a Stateless Session Beanek webszolgáltatásként is megjeleníthetőek.

**Locale és Remote interfészek:**

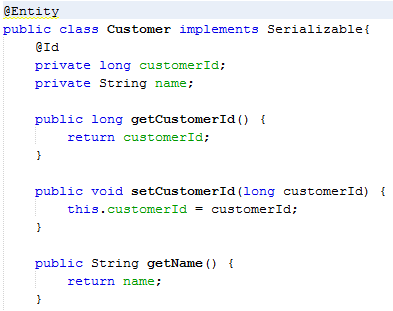
Locale interfészt használnuk általában, ha ejb-k közötti kommunikációt akarunk megvalósítani. Remote interfészt pedig, ha távoli ejb elérésre van szükség. Ezért is szokott szátválasztva lenni ez a két interfész, sőt ugyan azt a funkciót mindkét interfészbe nem is illik belerakni.

**Entitások és Java Perzisztencia API**

Ebben a fejezetben entitásokról lesz szó, amikről azt érdemes tudni, hogy adatbázis táblát objektum szinten tárolnak. Amíg a Session Beanek szolgáltatásokat biztosítanak a klienseknek, addig az entitások üzleti adatot reprezentálnak. Ezek is POJO-k mint a Session Beanek, és metaadat hozzáadásával (annotáció, xml) válik belőle Entitás.

**Entity:** Ha a bor webshop példát nézzük akkor a customer miként képzelhető el, Beanként vagy Entitásként?

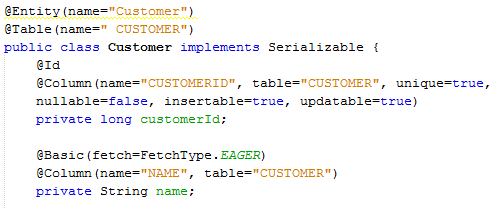
Természetesen entitásként, semmi funkciót nem lát el az osztály azon kívül, hogy vásárló adatokat tárol.



@Entity annotációval látjuk el azokat az osztályokat, amelyeknek az adatbázisban van tábla megfelelőjük. Az adattagokhoz a programming by default elvet követve nem kötelező a @Column annotáció, az alapból ott van, viszont a kulcs mezőhöz a @Id annotációt hozzá kell tennünk.

Egy Entity osztálynak szerializálhatónak kell lennie, az esetleges remote elérések miatt.

**Configuration by Default:** Szinte mindenre van alap annotáció, amiket ha nem jelzünk, akkor is ott vannak. Csak azokat az annotációkat kell kiírni, amelyek nem a default viselkedést írják le.



A fenti példakódban a default beállításoktól térünk el egy kicsit a vásárló entitásunkban.

@Basic annotáció: A primitív típusok (integer, string) automatikusan megkapják a JPA-tól a @Basic annotációt.

Típusok, melyek automatikusan megkaphatják a @Basic annotációt:

* Java primitívek (int, double, long, char, stb..)
* Java primitív csomagoló osztályok (Integer, Double, stb…)
* Szerializálható osztályok
* Felhasználó által definiált szerializálható osztályok
* Enumok
* java.util.Date
* java.util.Calendar
* java.lang.String.
* java.sql.Date
* java.sql.Time
* byte[]
* Byte[]
* char[]

@Transient annotáció: Azokat az adattagokat, amelyeket nem szeretnénk perzisztáltatni, azokat ezzel az annotációval kell ellátnunk (ilyen lehet például olyan érték, ami kiszámolható a többiekből, így a tárolása teljesen felesleges).

**Entity adat hozzáférés:** Itt arról van szó, ha az adattagokat annotálom, akkor a a Perzisztencia Menedzser közvetlenül írja és olvassa az adattagokat. Amikor nem az adattagokat annotálom, hanem a getter/setter metódusok egyikét, akkor a Perzisztencia Menedzser az írást és az olvasást ezeken a metódusokon keresztül végzi. Legyünk óvatosak, mert ha a getter/setter metódusok valamelyikébe üzleti logikát is teszünk, akkor az a perzisztáláskor is meghívódik a menedzser által, így mellékhatások keletkezhetnek. Az adattag annotáció előnye, hogy a getter/setter metódusok csak a kliens által hívódnak meg, a menedzser nem hívja ezeket.

**Elsődleges kulcsok:** Minden entitásnak (@Entity) kell, hogy legyen elsődleges kulcsa (@Id). A primary key lehet primitív típus (String, int, stb..), viszont lehet összetett típus is.

Egy egyszerű Primary Key adattag az @Id annotációval ellátva kész is van, nincs vele több dolog.

**Message-Driven Bean**

Üzenet orientált architektúra: Ez megvan céges anyagomban.

**JMS:** Java üzenet-orientált middleware API, ami alkalmazásoknak aszinkron üzenetek küldését és fogadását teszi lehetővé.

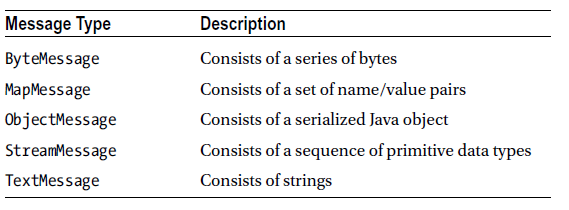
JMS architektúra részei:

* JMS provider: JMS szerverként is ismert.
* JMS kliens: Bármilyen java alkalmazás, vagy Java EE komponens, ami a JMS API-t használva fogad, vagy küld JMS üzeneteket.
* JMS gyártó(producer): Ő generál üzeneteket.
* JMS fogyasztó(consumer): Ő kapja meg az üzeneteket.
* JMS üzenet: Három részből áll, egy headerből, egy propertiesből, és body-ból. A header azonosítja az üzenetet, a properties olyan attribútumok, amelyek az alkalmazásnak kellenek, a body pedig az üzenet content.

A JMS API a következő szolgáltatásokat nyújtja:

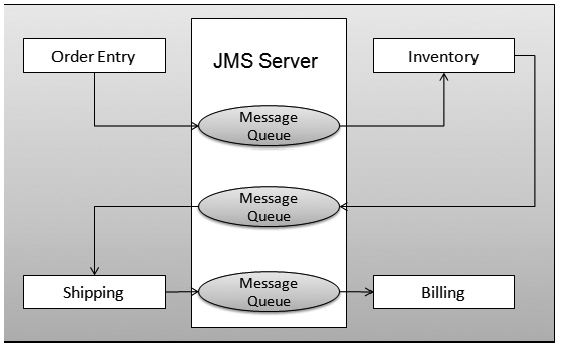
* elkészít
* küld
* kap
* üzenetet olvas

A JMS specifikáció többféle üzenettípust támogat.



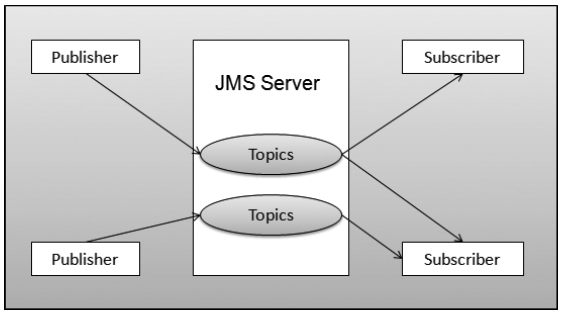
**Üzenetváltás architektúrái:**

* Point-to-point (p2p): Üzenet sorokra (message queues) épít, ahol a sorok tartalmazzák az üzeneteket, amelyeket a JMS kliens küldött. Az üzenet gyártó és fogyasztó megegyeznek egy közös queueban, amin keresztül váltanak üzeneteket. P2P modellt akkor alkalmazunk, ha egyetlen fogyasztó van minden üzenetre.



A fenti példán látható egy P2P modell, amiben minden Message Queue-ra egyetlen fogyasztó jut.

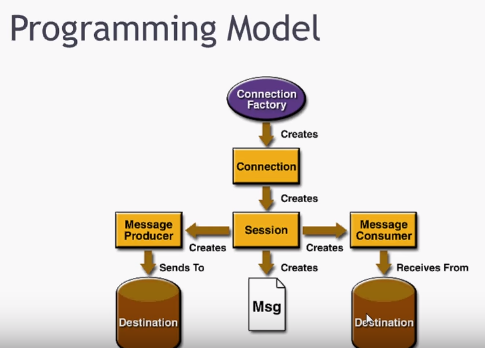
* Publish-subscribe (pub-sub) (közzétesz-feliratkozik): Ez a modell topikokra épít, ahol a topic a cél címe az üzeneteknek. Több feliratkozó, vagy JMS fogyasztó kaphatja meg az üzenetet.



**Message-Driven Bean használata:** Ez egy aszinkron üzenet fogyasztó(consumer), ami üzeneteket dolgoz fel a JMS-en keresztül. MDB(Message-Driven Bean) végzi az üzenetek feldolgozását, az EJB konténer, amiben az MDB-k futnak nyújtja ezen esetben is a szolgáltatásokat, amiket a többi bean esetében is(tranzakciókezelés, konkurenciakezelés, stb..), a bean fejlesztőnek így csak az üzleti logika implementálásával kell foglalkoznia. Az MDB-k természetükből adódóan Stateless-ek. Az EJB kliensek nem hívhatnak rá közvetlenül az MDB metódusaira, az egyetlen mód, hogy kommunikálni tudjanak velük, hogy JMS üzenetet küldenek oda, amit az MDB figyel. Bármely java alkalmazás vagy Java EE komponens, ami JMS-t használ, lehet üzenet küldő az MDB felé, ha üzenetet küld a queue-ra vagy topicra.

**JMS üzenetek feldolgozása lehet szinkron és aszinkron is:**

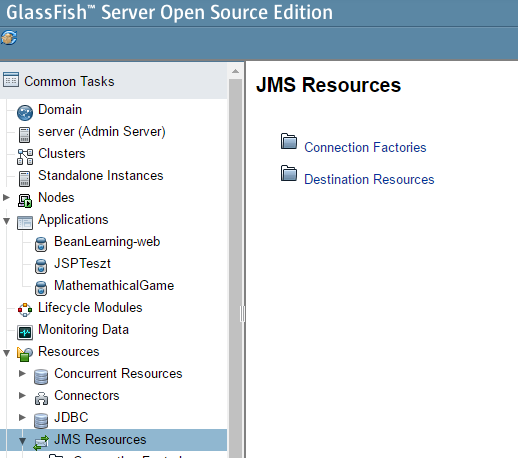
* Szinkron: receive() metódus segítségével
* Aszinkron: Message Listener interfészen keresztül



A message producer feladata az üzenetek elkészítése.

A message consumer kapja meg az üzeneteket.

**JMS beállítása NetBeans ide-ben:**



Ha elindítottuk a GlassFish-t, akkor a localhostra a 4848-as porton behozhatjuk az admin felületet, ami a fenti képen is látszik, és itt beállíthatunk JMS erőforrásokat.