K&A

Auf dem Weg zu Enterprise Java Beans - Ein Erfahrungsbericht

Enterprise Java Beans (EJB) ist Stand der Technik bei der Entwicklung unternehmensweiter - und darüber hinausgehender - Anwendungen.

Nachdem

- sich Java in den letzten Jahren als wichtigste objektorientierte Programmiersprache durchgesetzt hat und sich aktuell auch die Welt der Server erobert,
- sich Architekturen als besonders wartungs- und erweiterungsfreundlich erweisen, wenn sie in Komponententechnologie entwickelt sind,
- die Tendenz zum Einsatz von n-Tier-Architekturen (Präsentation, Geschäftslogik & Persistenz) bei Internet-Anwendungen unverkennbar ist,

war die Entwicklung eines entsprechenden Frameworks (J2EE von SUN) folgerichtig.

EJBs sind - als wesentlicher Teil der J2EE - in weiten Teilen spezifiziert und inzwischen ein anerkannter Standard. EJBs können jedoch nicht isoliert betrachtet werden, sondern müssen stets in den Gesamtkontext der Anwendungsarchitektur eingeordnet werden:

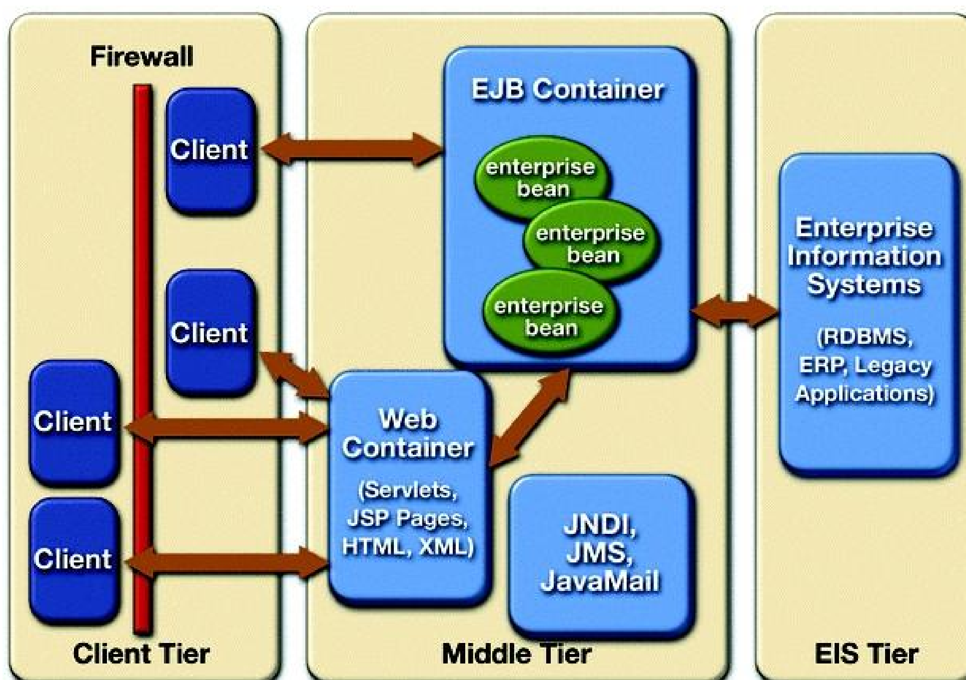


Abbildung: N-Tier-Architektur (Quelle: Sun Microsystems)

© Kölsch & Altmann GmbH, 2005

Frameworks im Allgemeinen und das J2EE-Framework im Speziellen haben einige wesentliche Vorteile:

- Die ins Framework eingeflossenen Erfahrungen (und Fehler) müssen nicht nochmals gemacht werden, das Rad muss nicht immer wieder neu erfunden werden.
- Weit verbreitete Frameworks haben den Vorteil, dass sie von vielen Nutzern eingesetzt und verifiziert werden. Dadurch ist schon allein die Testabdeckung durch die Benutzer extrem hoch. Werden Fehler nicht selbst gefunden, sind sie höchstwahrscheinlich schon von jemand anderem gefunden worden.
- Durch das Framework wird eine Architektur vorgegeben, die sich in den Köpfen der Beteiligten manifestieren kann. Hierdurch lässt sich die volle Konzentration auf die Geschäftslogik legen.
- Erweiterungen bzgl. des Frameworks müssen – möglicherweise – nicht in allen Applikationen nachgezogen werden, sondern nur an zentraler Stelle, z.B. beim Application Server.

J2EE definiert eine Plattform für die Entwicklung von verteilten, objektorientierten Anwendungen, die insbesondere in unternehmenskritischen Bereichen eingesetzt werden. Dabei hat sich die Gliederung in mehrere Schichten (Client-Tier, Web-Tier, Application- bzw. Middle-Tier und EIS-Tier) als besonders vorteilhaft herausgestellt, weil hierdurch zum einen die einzelnen Teile getrennt implementiert werden können, zum anderen aber auch eine hohe Skalierbarkeit unterstützt wird.

Die Geschäftslogik kann vollständig von der Präsentationsschicht abgeschirmt werden. Des Weiteren werden dem Entwickler Aufgaben bzgl. Transaktionsmanagement, Nebenläufigkeit, Zugriffskontrolle, verteilte Ressourcenverwaltung im Wesentlich abgenommen bzw. erheblich erleichtert.

© Kölsch & Altmann GmbH, 2005

Für das Beratungshaus Kölsch & Altmann GmbH ist es immer eine wichtige Aufgabe, sich mit neuen Technologien auseinanderzusetzen und ihre Einsatzfähigkeit zu beurteilen. Dieses Projekt diente dazu, sich in die Thematik J2EE und EJB einzuarbeiten und die Ergebnisse auch anderen interessierten Einsteigern zur Verfügung zu stellen.

Nun gibt es neben den verschiedenen Spezifikationen der J2EE eine ganze Reihe von Publikationen zu diesem Thema. Unser Fokus hier soll keine Erweiterung bzw. Wiederholung der Spezifikation sein, sondern wir haben einzelne Aspekte im Zusammenhang herausgearbeitet und unsere Erfahrungen im Umgang mit J2EE mit einfließen lassen.

Dieses Dokument

- zeichnet verschiedene Erfahrungen auf,
- geht hierbei im Speziellen auf die Umsetzung und das Deployment auf verschiedenen Application-Servern ein:
    - Referenzimplementierung von SUN (Sun-RI),
    - BEA Web-Logic-Server (WLS),
    - JBoss
- behandelt verschiedene Aspekte von „unternehmensweiten Anwendungen“ wie
    - die Anbindung von Web-Servern via Servlets bzw. Java Server Pages (JSP),
    - den Java Naming & Directory Service (JNDI),
    - den Java Message Service (JMS),
    - die Java Transaction API (JTA)
    - und „altbekannte“ Themen wie JDBC und JavaMail.

Im Folgenden einige Ausarbeitungen von interessanten Aspekten von J2EE:

- Transaktionen & Isolation-Level
- JNDI & EJB-Referenzen
- Java Message Service im J2EE-Kontext
- Primary & Foreign Keys » CMR

Von einem – allerersten – Projekt mit der J2EE-Technologie (Online-Seminar-Verwaltung (OSV)) stammen die auf diesen Seiten aufgeführten Beispiele. Dieses Projekt war ursprünglich aufgesetzt worden mit dem Ziel, einerseits mit der Technologie vertraut zu werden, das Ergebnis langfristig aber andererseits auch für die Seminarverwaltung der Firma produktiv nutzen zu können. Ganz bewusst wurden hierbei alle möglichen Arten und Ausprägungen von Enterprise Java Beans eingesetzt, um eine möglichst hohe Evaluationsabdeckung zu erreichen.

© Kölsch & Altmann GmbH, 2005

Online-Seminar-Verwaltung (OSV)

Inhalt

- Projektdefinition und Vorgaben
- Auszüge Requirements-Modell
- Auszüge Design
- Auszüge Implementierung

Projektdefinition und Vorgaben

Aufgesetzt wurde das Projekt OSV ursprünglich noch i. Zshg. mit der EJB-Spezifikation V1.1. Ziel des Projektes sollte ein Softwaresystem sein, mit dessen Hilfe es mittelfristig möglich ist,

§ *- in der Rolle als Administrator –*
Seminare im Netz (zunächst Intra-, später Internet) anzubieten,

- *- in der Rolle als Kunde (bzw. als Mitarbeiter i. Zshg. mit der Teilnahme an einem haus-internen Seminar)* -
Seminare (für evtl. auch mehrere Teilnehmer) zu buchen bzw. auch wieder zu stornieren.

Umzusetzen war das zu entwickelnde System mit der üblichen J2EE-Architektur, wobei die Servlet-Anteile nach Benutzerrollen getrennt paketiert sein sollten:

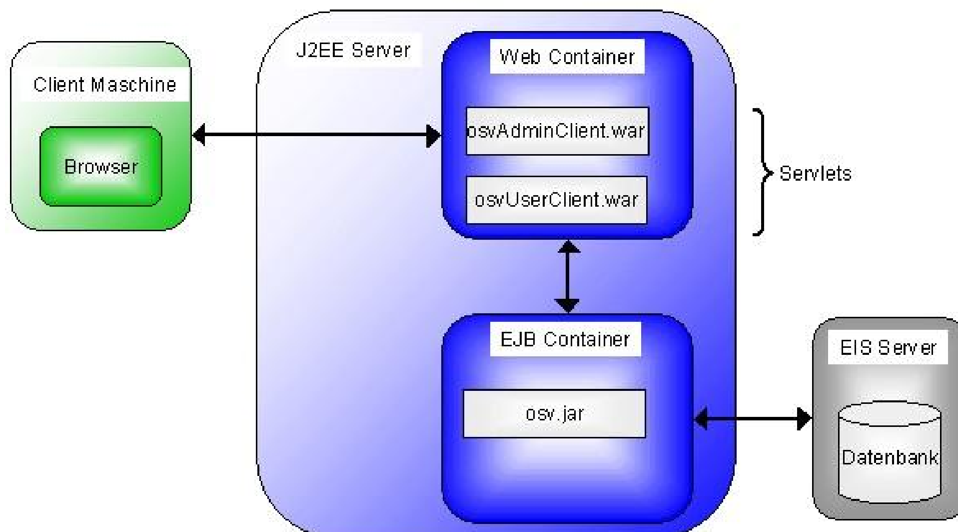


Abbildung: OSV-Architektur

© Kölsch & Altmann GmbH, 2005

Einiges an Systemfunktionalität findet in mehreren Paketen Wiederverwendung. Die erstellten Pakete sollten aber immer abgeschlossen sein, laufen sie möglicherweise doch letztlich in verschiedenen Containern.

Beim ersten Ansatz spielte die Oberfläche des Clients nur in Bezug auf seine Funktionalität eine Rolle. Zum Einsatz kamen Servlets. In einer zweiten Phase sollte dann auf JSP (Java-Server-Pages) umgestellt werden.

Ein solches System ist nur in hohem Maße nebenläufig, wenn eine große Anzahl von Seminaren angeboten und - v. a. auch parallel - gebucht werden. Hohe Anforderungen an die Nebenläufigkeit waren also zunächst nicht gegeben. Nichtsdestoweniger sollte es z. B. nicht möglich sein, dass zwei Kunden mehr oder weniger gleichzeitig erfolgreich den letzten freien Platz in einem Seminar buchen können.

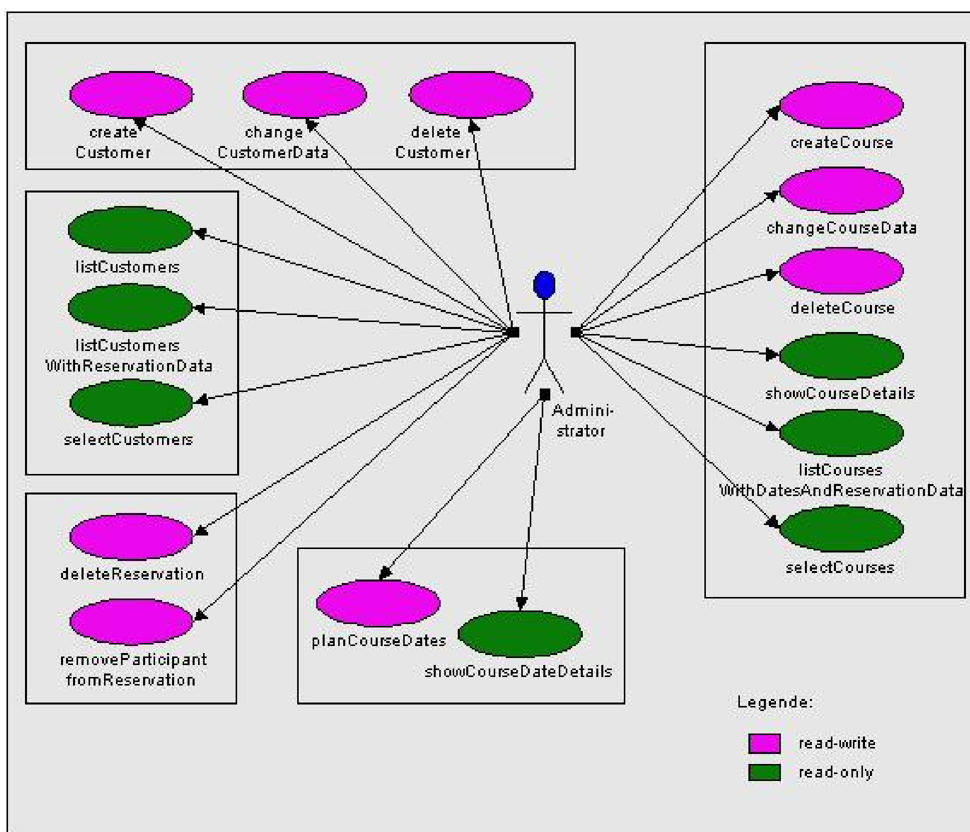
EJB ist auch nicht eine Technologie, die nur dann einzusetzen ist, wenn hohe Anforderungen an die Nebenläufigkeit spezifiziert sind, sondern ist auch ein Entwicklungs-Framework, mit dessen Hilfe

- Systeme immer wieder in eine allseits akzeptierte – und inzwischen evaluierte und bekannte – Form gegossen werden,
- der Entwickler v. a. beim Einsatz von CMP (Container Managed Persistence) bei der Initialisierung und Verwendung von Ressourcen entlastet wird, indem er diese nur via in JNDI (Java Naming & Directory Service) definierter Symbole anspricht, was den Vorteil hat, dass zum Zeitpunkt der Entwicklung noch gar nicht bekannt sein muss, auf welcher Plattform das System schlussendlich zum Einsatz kommen wird bzw. welcher konkreten Ressourcen sich bedient wird.

© Kölsch & Altmann GmbH, 2005

Auszüge Requirements-Modell

Die folgenden Diagramme zeigen die verschiedenen Use Cases, die beim Einrichten einer Online-Seminar-Verwaltung bzgl. des Auswählens, Buchens und Stornierens von Kursen relevant sind, in einer ersten Sicht die Use Cases, die einem Administrator zur Verfügung stehen müssen:



Der Administrator muss hierbei - wie zu ersehen - folgende grundlegende Basisfunktionen durchführen können:

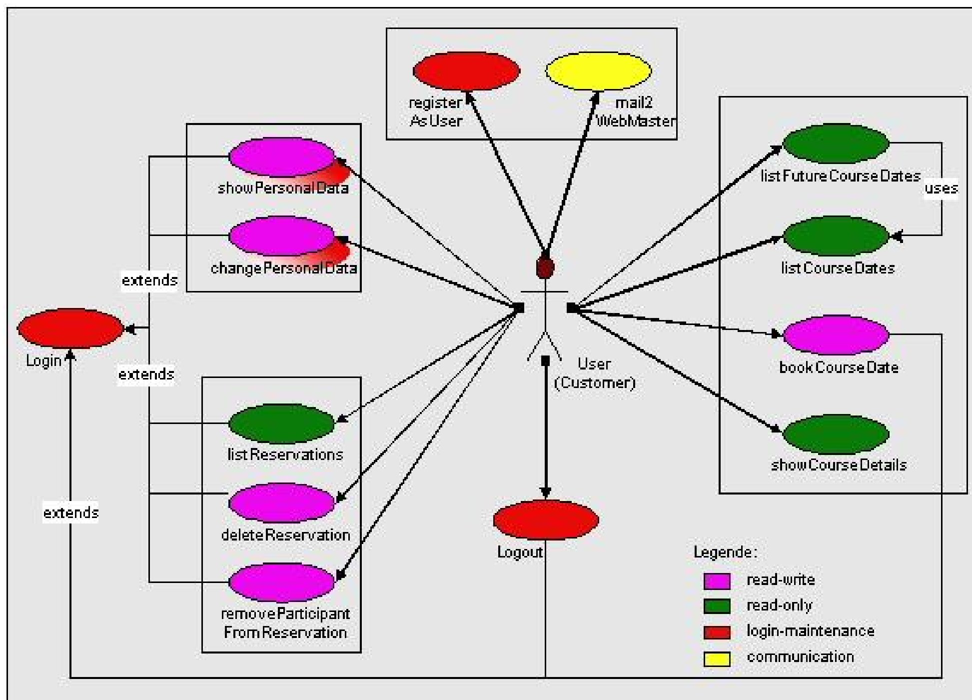
- Verwalten von Kunden
- Verwalten von Kursen
- Verwalten von Kursterminen
- Anzeigen von Kunden und Auswählen von Kunden nach bestimmten Kriterien
- Reservierungen korrigieren

Hierbei ist auch schon die – farbliche – Unterscheidung getroffen, welche Daten dabei aus der DB nur gelesen und welche auch geändert werden. Des Weiteren immer auch darauf zu achten, dass auch mehrere Benutzer des Systems miteinander konkurrieren, zwar noch recht unwahrscheinlich bei der Rolle "*Administrator*", jedoch schon sehr viel wahrscheinlicher bei der Rolle "*Customer*". Hier ist dann eine geeignete Serialisierung der Anfragen an das System zu

© Kölsch & Altmann GmbH, 2005

gewährleisten. Siehe hierzu auch Transaktionen und **Isolation-Level**.

Für die Zugriffe von Kunden stehen folgende Use Cases zur Verfügung:



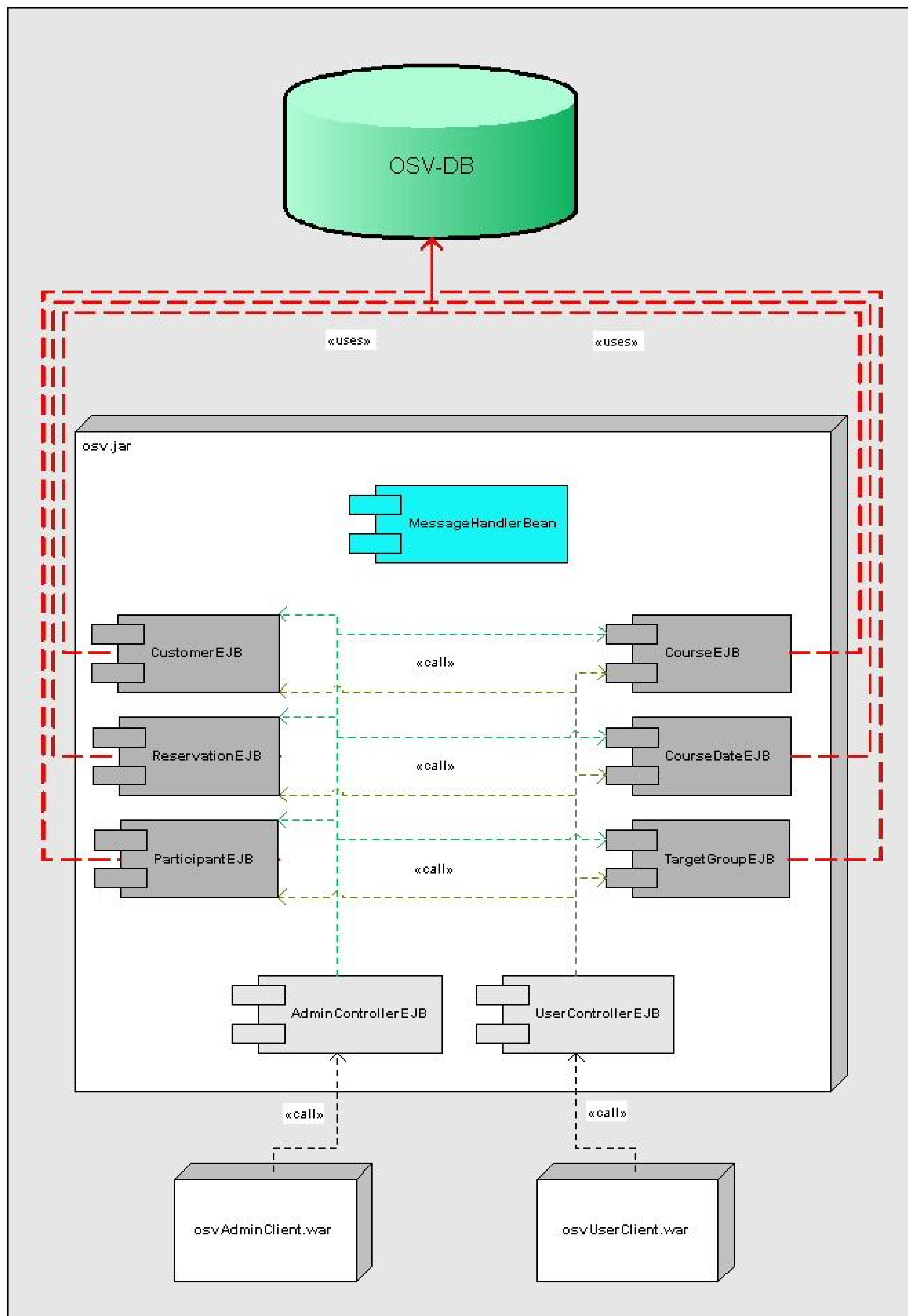
Der Kunde muss hierbei - wie zu ersehen - folgende grundlegende Basisfunktionen durchführen können:

- Anzeigen und Auswählen von Seminaren und Seminarterminen
- Registrierung als neuer Kunde, Aufbau einer Login-Session
- Verwalten von (*seinen*) Kundendaten
- Reservieren und Stornieren von Seminaren zu einem bestimmten Termin für ausgewählte Teilnehmer seines Hauses

© Kölsch & Altmann GmbH, 2005

Auszüge Design

Im Folgenden zunächst ein Diagramm, das die Server-seitige Komponentenstruktur der OSV aufzeigt:



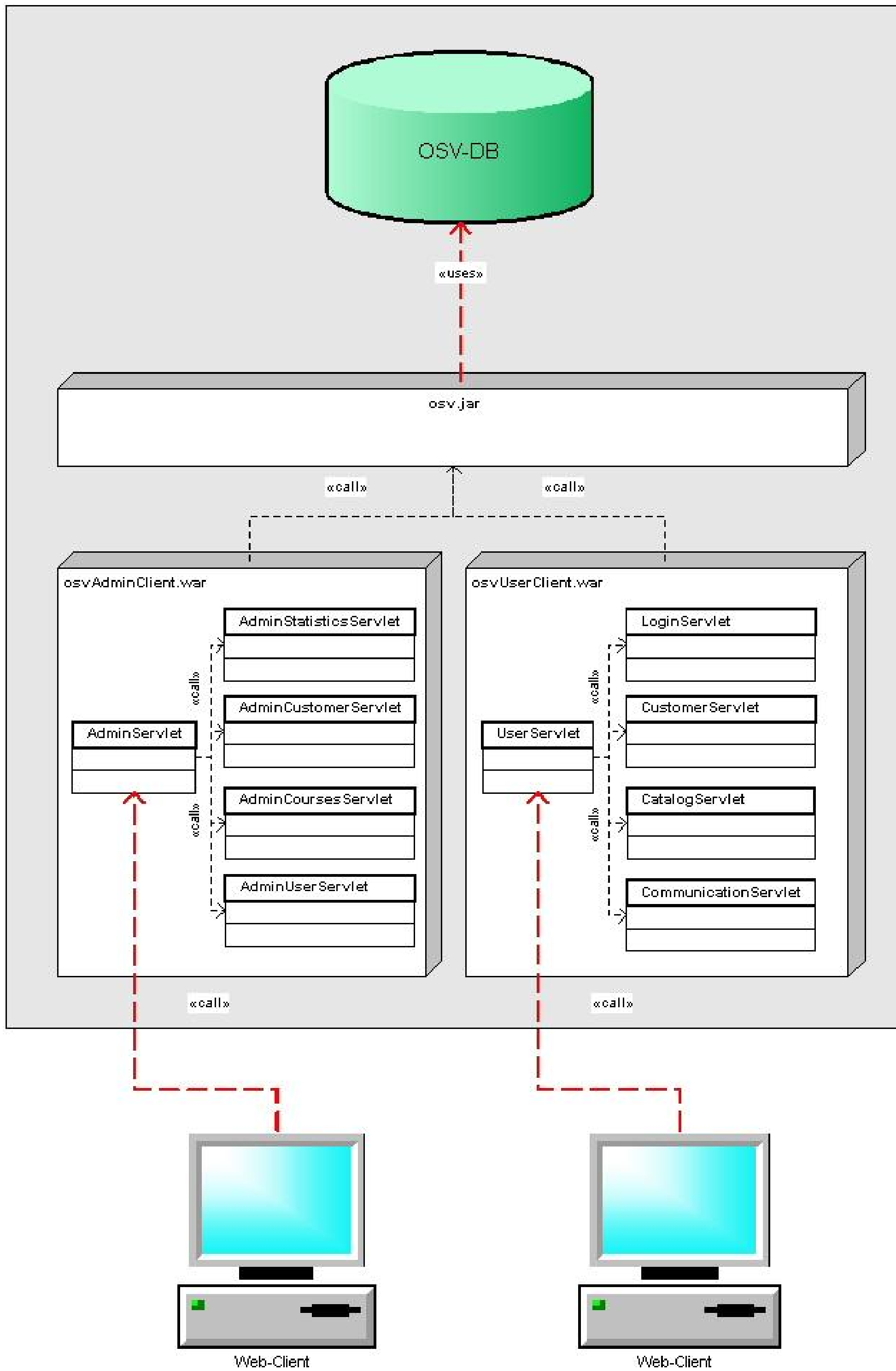
© Kölsch & Altmann GmbH, 2005

Ins Auge fällt zunächst, dass nur zwei Komponenten von außen zugreifbar sind, "*AdminControllerEJB*" und "*UserControllerEJB*". Es handelt sich dabei jeweils um ein *stateful* Session-Bean. Die Architektur genügt damit dem *Facade*-Design-Pattern. Des Weiteren ist dieses J2EE-Design-Pattern in aller Regel gepaart mit dem Pattern der *Data-Transfer-Objekts*. Dieses stellt sicher, dass die in den WAR-Dateien enthaltenen Servlets, die keinen Zugriff auf Entity-Beans haben, geeignete Klassen zum Austausch der Informationen mit dem Admin- bzw. UserControllerEJB zur Verfügung gestellt bekommen müssen. Diese Klassen beinhalten – den Attributen der Entity-Beans – entsprechende Attribute, sind aber selbst nur einfache Bean-Klassen. Aufgabe der Controller ist es, die Informationen aus diesen einfachen Bean-Objekte mittels der entsprechenden Entity-Beans persistent zu machen.

Des Weiteren fällt auf, dass die Message-Driven-Bean keine Verbindungen zu anderen Beans besitzt. Das muss nicht so sein, liegt im Zusammenhang mit der OSV aber lediglich daran, dass die Bean nur zu Logging-, Tracing- und Email-Zwecken verwendet wird. Zu diesem Zweck benötigt sie keine Hilfe von anderen Beans. Getriggert wird sie selbst durch entsprechende Einträge in die ihr zugeordnete Message-Queue. Bzgl. der zugrunde liegenden Mechanismen siehe Java Message Service im J2EE-Kontext.

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein Diagramm, das die Servlet-Struktur im Web-Container aufzeigt:



© Kölsch & Altmann GmbH, 2005

Hier kommen ebenfalls die Prinzipien der strukturierten Programmierung zum Einsatz in dem Sinne, dass nicht alle Logik in einem Servlet realisiert ist, sondern themenbezogen verteilt über jeweils mehrere Servlets. Der Einstieg von Seiten des Web-Clients stellt jedoch ein zentrales Servlet dar, das *AdminServlet* für die Rolle als Administrator, das *UserServlet* für die Rolle als Kunde.

Nichtsdestoweniger sind Servlets allein nicht der Weisheit letzter Schluss, ein wesentlicher Fortschritt gegenüber CGI zwar, aber nur im ersten Ansatz sinnvoll, weil portabel und unbegrenzt erweiterbar. Das Problem, das sich beim Einsatz von Servlets darstellt, ist die Unübersichtlichkeit durch eine endlose Folge von *println()*-Kommandos für den Aufbau der HTML-Seite. Die Lösung hierfür sind *Java Server Pages (JSP)*, womit sich eine HTML-Seite direkt aufbauen lässt und an den dynamischen Stellen Java-Code aufgerufen werden kann. Das Problem bei der Verwendung von JSP allein ist jedoch die fehlende Möglichkeit, den Kontrollfluss abzubilden. Hierfür müssen neben Java Server Pages letztlich doch wieder Servlets eingesetzt werden. Damit folgt dieses Konzept dem von Smalltalk her bekannten *Model-View-Controller*-Design-Pattern.

Eine Umsetzung des MVC-Pattern findet im Open-Source-Framework Struts (Jakarta - Apache) folgendermaßen statt:

- ***Model***:
  Das Modell teilt sich auf in den Zustand des Systems und die möglichen Aktionen, um diesen zu verändern. Der Systemzustand ist dabei durch die – im EJB-Container angesiedelte und durch das *Action-Servlet* (s.u.) in den Web-Container replizierte – Business-Logik abgebildet.
  Um den Systemzustand im Web-Container abzubilden, kommen JavaBeans zum Einsatz, die einerseits die mittels Java Server Pages auszugebenden Daten, andererseits auch die von Web-Clients auf HTML-Forms eingegebenen Daten beinhalten.

- ***View***:
  Die View wird rein äußerlich durch den Browser repräsentiert, der den intern von Java Server Pages generiertem HTML-Code anzeigt und entsprechende HTTP-Requests wieder zurücksendet, die wiederum vom *Action-Servlet* (s.u.) entgegengenommen werden.

- ***Controller***:
  - Das *Action-Servlet* nimmt HTTP-Request entgegen und gibt den Kontrollfluss an Hand eines konfigurierten Action-Mappings weiter an entsprechende *Action-Objekte*.
  - *Action-Objekte* sind – als Wrapper-Klassen für die Business-Logik – für die Abhandlung der enstprechenden Aktionen verantwortlich, die der HTML-Client durch eine bestimmte Aktion auslöst. Action-Objekte haben gleichzeitig Zugriff auf das *Action-Servlet* der Web-Applikation, denn nicht zuletzt haben Action-Objekte auch die Aufgabe – nach Abhandlung der durchzuführenden Aktion – den Kontrollfluss – über das *Action-Servlet* – an eine andere Komponente weiterzugeben, z.B. an eine Java Server Page zur Ausgabe des Ergebnisses.

© Kölsch & Altmann GmbH, 2005

Auszüge Implementierung

Servlets

Servlets sind der – im Web-Server laufende –- Softwareanteil einer unternehmensweiten Anwendung, der die Kommunikation zwischen einem Browser oder anderen HTTP-Clients und dem HTTP-Server sicherstellt. Die Aufgaben von Servlets sind

- die Auswertung der vom Benutzer eingegebenen bzw. vom Browser übermittelten Daten,
- die Generierung der an den HTTP-Client zurückzusendenden Daten (in der Regel in HTML) und
- das Senden des generierten Dokuments an den HTTP-Client.

Ein Servlet ist eine von der abstrakten Klasse *HttpServlet* abgeleitete Java-Klasse. Für das Format eines eingehenden Requests bieten sich – überwiegend – zwei Formate an: *GET* und *POST*. Abhängig vom verwendeten Format werden die entsprechenden Methoden *doGet* bzw. *doPost* aufgerufen, die gerne auf eine gemeinsam aufgerufene Methode abgebildet werden, weil sich die Requests semantisch nicht wesentlich unterscheiden. Die Unterschiede sind im Aufbau der Request-Nachricht begründet:

- Bei einem GET-Request werden die Parameter mit im Request-Header übertragen, ein Body ist nicht vorhanden.
- Bei einem POST-Request werden die Parameter im Request-Body übertragen.

Zweiteres ist zum einen sicherer, weil sensible Daten nicht Teil der angezeigten URL sind. Zum anderen ist die URL in ihrer Länge beschränkt, mit zuvielen Parametern würde diese Grenze überschritten.

Servlets können auch strukturiert werden. Zunächst einmal ein Beispiel, das die Strukturierung in mehrere Servlets dokumentiert. Weitere Servlets werden quasi inkludiert: Dabei werden der Request- und der Response-Parameter einfach durchgereicht. Ein "Unter"-Servlet kann also letztlich ein Stück HTML-Code an das schon vom "Haupt"-Servlet in Bearbeitung befindliche HTML-Dokument anfügen und dieses erweiterte HTML-Dokument dann wieder an den Aufrufer zur weiteren Bearbeitung zurückgeben.

© Kölsch & Altmann GmbH, 2005

Im folgenden Auszug baut das *AdminServlet* den HTML-Rahmen, das *AdminCustomerServlet* ist für die Kodierung des HTML-Body zuständig:

```
public class AdminServlet extends HttpServlet {

    public void init(ServletConfig config) throws
                                                        ServletException
    {
            super.init(config);
    }

    public void destroy() {}

    public String getServletInfo() {
      return "Servlet for the administrator of the OSV";
    }

    protected void processRequest(HttpServletRequest  request,
                                  HttpServletResponse response)
      throws ServletException, IOException
    {
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter();

      RequestDispatcher adminCustomersServlet =
        getServletContext().getRequestDispatcher
                                       ("/adminCustomersServlet");

      out.println
        ("<html><head><title>Admin-Servlet</title></head><body>");

      adminCustomersServlet.include(request, response);

      out.println(("</body>>");
      out.close();
    }

    protected void doGet(HttpServletRequest  request,
                         HttpServletResponse response)
      throws ServletException, IOException
    {
       processRequest(request, response);
    }

    protected void doPost(HttpServletRequest  request,
                          HttpServletResponse response)
      throws ServletException, java.io.IOException
    {
              processRequest(request, response);
    }
  }
```

© Kölsch & Altmann GmbH, 2005

K&A

Beans

Bei der Implementierung der Beans wird hat man die Alternative zwischen *Container Managed Persistence (CMP)* und *Bean Managed Persistence (BMP)*, die sich bzgl. des Aufwands der Implementierung deutlich unterscheiden. Bei Container Managed Persistence ist der DB-Zugriff rein deskriptiv (in verschiedenen XML-Deskriptoren) zu beschreiben, während bei Bean Managed Persistence die Zugriffslogik selbst implementiert werden muss. Dem deskriptiven Konzept sind allerdings Schranken auferlegt, die nicht zuletzt mit der in der evaluierten Version EJB 1.1 noch etwas rudimentären EJB-Query-Language zusammenhängen. Genauso sind Container-Managed-Relationships (CMR) bzgl. der Performanz mit Vorsicht einzusetzen, dies v.a., wenn die referenzierten Tabellen etwas größer sind.

Auf die Kodierung der Beans wollen wir im einzelnen nicht eingehen, viel interessanter sind die Zusammenhänge, die in den verschiedenen plattformunabhängigen und -abhängigen XML-Deskriptoren modelliert werden. Denn das ist das wirklich Neue und Aufregende an der EJB-Technologie.

XML-Deskriptoren

Zur Erinnerung an dieser Stelle noch einmal die wesentlichen Gründe, warum es Sinn macht, Implementierung und Deployment zu trennen:

- Portabilität des Source-Code
  (Portierung auf verschiedene Ablaufumgebungen ohne Anpassung des Source-Code!)
- Bean-Entwickler muss Ablaufumgebung nicht kennen
- Beschreibung der Software-Komponenten durch den Deskriptor

Unterschieden werden die Deskriptoren der Web-Applikation und die der EJB-Komponenten, darunter jeweils plattformunabhängige und -abhängige Deskriptoren:

- Web-Applikation
  - web.xml (plattformunabhängig)
  - weblogic.xml (plattformabhängig: BEA Weblogic)
- EJB-Komponenten
  - ejb-jar.xml (plattformunabhängig)
  - weblogic-ejb-jar.xml (plattformabhängig: BEA Weblogic)
  - weblogic-cmp-rdbms-jar.xml (plattform- und DBMS-abhängig: BEA Weblogic / Oracle)

Web: "web.xml"

Diese Datei stellt die plattformunabhängigen Einstellungen bzgl. einer Web-Applikation dar, beschreibt also einen Web-Client bzw. sein zugehöriges war-Archiv. Hierzu gehören Aspekte wie

- die Auflistung aller zugehörigen Servlets
- Sicherheit bzgl. der Zugriffskontrolle

© Kölsch & Altmann GmbH, 2005

- von den Servlets referenzierte Beans

Die Implementierung dieses Deskriptors liegt in der Verantwortung des Web-Applikationsentwicklers.

Im Folgenden ein beispielhafter Auszug eines entsprechenden Deskriptors:

```
<web-app>

  <servlet>
    <servlet-name>UserServlet</servlet-name>
    <servlet-class>
      de.kamuc.osv.client.servlet.UserServlet
    </servlet-class>
  </servlet>

  <security-role>
    <description>
       The role allowed to access our content
    </description>
    <role-name>guest</role-name>
  </security-role>

  <ejb-ref>
    <ejb-ref-name>ejb/codedNameUserController</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>de.kamuc.osv.ejb.controller.UserControllerHome</home>
    <remote>de.kamuc.osv.ejb.controller.UserController</remote>
  </ejb-ref>

</web-app>
```

Web: "weblogic.xml"

Diese Datei stellt die plattformabhängigen Einstellungen bzgl. einer Web-Applikation dar, beschreibt also einen Web-Client bzw. sein zugehöriges war-Archiv bzgl. der plattformabhängigen Einstellungen. Hierzu gehören Aspekte wie

- die Konfiguration diverser Web-Client-Parameter (z.B. Timeout)
- Auflösung der Bean-Referenzen (Vergabe von JNDI-Namen)

Die Implementierung dieses Deskriptors liegt in der Verantwortung des Web-Deployer.

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein beispielhafter Auszug eines entsprechenden Deskriptors (BEA WebLogic):

```
<weblogic-web-app >

  <session-descriptor>
    <session-param>
      <param-name>TimeoutSecs</param-name>
      <param-value>3600</param-value>
    </session-param>
  </session-descriptor>

    ...

  <reference-descriptor>
    <ejb-reference-description>
      <ejb-ref-name>
         ejb/codedNameUserController
      </ejb-ref-name>
      <jndi-name>JNDINameBean2</jndi-name>
    </ejb-reference-description>
  </reference-descriptor>

</weblogic-web-app>
```

EJB: *"ejb-jar.xml*"

Diese Datei stellt die plattformunabhängigen Einstellungen bzgl. einer EJB-Applikation dar, beschreibt also eine Menge von Komponenten bzw. deren zugehöriges jar-Archiv. Hierzu gehören Aspekte wie

- Namen und Art der enthaltenen Beans
- die Attribute und Beziehungen von Entity Beans
- innerhalb des Source-Code referenzierte Ressourcen (Umgebungsvariablen, andere Beans, JMS-Destinations, ...)

Die Implementierung dieses Deskriptors liegt in der Verantwortung des Bean-Providers, vervollständigt evtl. durch den Application Assembler.

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein beispielhafter Auszug eines entsprechenden Deskriptors:

```
<ejb-jar>

  <!-- Beans -->

  <enterprise-beans>
    <session>
      <ejb-name>UserControllerEJB</ejb-name>
      <home>de.kamuc.osv.ejb.controller.UserControllerHome</home>
      <remote>de.kamuc.osv.ejb.controller.UserController</remote>
      <ejb-class>
         de.kamuc.osv.ejb.controller.UserControllerEJB
      </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-local-ref>
        <ejb-ref-name>ejb/codedNameCourse</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>de.kamuc.osv.ejb.CourseLocalHome</local-home>
        <local>de.kamuc.osv.ejb.course.CourseLocal</local>
        <ejb-link>CourseEJB</ejb-link>
      </ejb-local-ref>

    </session>
  </enterprise-beans>

  <!-- Beziehungen zwischen Beans -->

  <ejb-relation>
    <ejb-relation-name>
       CourseLocal-TargetGroupLocal
    </ejb-relation-name>

    <ejb-relationship-role>
      <ejb-relationship-role-name>
         CourseLocal-Has-TargetGroupLocal
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>CourseEJB</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>targetGroup</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>

    <ejb-relationship-role>
      <ejb-relationship-role-name>
         TargetGroupLocal-Has-CourseLocal
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
```

© Kölsch & Altmann GmbH, 2005

K&A

```
          <ejb-name>TargetGroupEJB</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>
  </ejb-jar>
```

EJB: "weblogic-ejb-jar.xml"

Diese Datei stellt die plattformabhängigen Einstellungen bzgl. einer EJB-Applikation dar, beschreibt also eine Menge von Komponenten bzw. deren zugehöriges jar-Archiv bzgl. der plattformabhängigen Einstellungen. Hierzu gehören Aspekte wie

- Auflösung der Referenzen (Vergabe von JNDI-Namen)
- Festlegungen bzgl. des zugrundeliegenden DBMS
- Festlegung von Isolation-Levels für die einzelnen Methoden

Die Implementierung dieses Deskriptors liegt in der Verantwortung des Deployer.

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein beispielhafter Auszug eines entsprechenden Deskriptors (BEA WebLogic):

```
<weblogic-ejb-jar>

  <!-- Session Beans -->

  <weblogic-enterprise-bean>
    <ejb-name>UserControllerEJB</ejb-name>

    <reference-descriptor>
      <resource-description>
        <res-ref-name>
           jms/codedNameConnectionFactory
        </res-ref-name>
        <jndi-name>JNDINameConnectionFactory</jndi-name>
      </resource-description>

      <resource-env-description>
        <res-env-ref-name>
           jms/codedNameQueue
        </res-env-ref-name>
        <jndi-name>JNDINameQueue</jndi-name>
      </resource-env-description>
    </reference-descriptor>

    <jndi-name>JNDINameBean2</jndi-name>

  </weblogic-enterprise-bean>

  ...

  <!-- Entity Beans -->

  <weblogic-enterprise-bean>
    <ejb-name>CourseEJB</ejb-name>
    <entity-descriptor>

      <persistence>
        <persistence-type>
          <type-identifier>
             WebLogic_CMP_RDBMS
          </type-identifier>
          <type-version>6.0</type-version>
          <type-storage>
             META-INF/weblogic-cmp-rdbms-jar.xml
          </type-storage>
        </persistence-type>

        <persistence-use>
          <type-identifier>
             WebLogic_CMP_RDBMS
          </type-identifier>
          <type-version>6.0</type-version>
        </persistence-use>
```

© Kölsch & Altmann GmbH, 2005

```
        </persistence>

      </entity-descriptor>

      <local-jndi-name>JNDINameBean4</local-jndi-name>

    </weblogic-enterprise-bean>

    ...

    <!-- Isolation Level -->

    <transaction-isolation>
      <isolation-level>
         TRANSACTION_READ_COMMITTED_FOR_UPDATE
         <method>
            <ejb-name>UserControllerEJB</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>reserveCourse</method-name>
            <method-params>
               <method-param>
                  de.kamuc.osv.ejb.reservation.ReservationData
               </method-param>
            </method-params>
         </method>
      </isolation-level>
    </transaction-isolation>
  </weblogic-ejb-jar>
```

EJB: "weblogic-cmp-rdbms-jar.xml"

Diese Datei stellt die plattformabhängigen Einstellungen bzgl. einer EJB-Applikation bezogen auf ein zugrundeliegende DBMS im Falle der Verwendung von CMP dar, beschreibt in diesem Kontext eine Menge von Komponenten bzw. deren zugehöriges jar-Archiv. Hierzu gehören Aspekte wie

- Mapping der Entity Beans auf die entsprechenden Tabellen
- Mapping der Entity-Bean-Attribute auf die Tabellenspalten
- Festlegungen der Fremdschlüsselspalten bei CMR

Die Implementierung dieses Deskriptors liegt in der Verantwortung des Deployer.

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein beispielhafter Auszug eines entsprechenden Deskriptors (BEA WebLogic):

```
<weblogic-rdbms-jar>

  <!-- Beans -->

  <weblogic-rdbms-bean>
    <ejb-name>CourseDateEJB</ejb-name>
    <data-source-name>JNDINameOracleOSV-DB</data-source-name>
    <table-name>coursedate</table-name>
    <field-map>
      <cmp-field>dateBegin</cmp-field>
      <dbms-column>courseDateBegin</dbms-column>
    </field-map>

            ...

    <automatic-key-generation>
      <generator-type>ORACLE</generator-type>
      <generator-name>OSV_seq</generator-name>
      <key-cache-size>10</key-cache-size>
    </automatic-key-generation>

  </weblogic-rdbms-bean>

  <!-- Beziehungen zwischen Beans -->

  <weblogic-rdbms-relation>
    <relation-name>
       CourseLocal-TargetGroupLocal
    </relation-name>
    <table-name>course_targetGroup_link_KD</table-name>
    <weblogic-relationship-role>
      <relationship-role-name>
         CourseLocal-Has-TargetGroupLocal
      </relationship-role-name>
      <column-map>
        <foreign-key-column>courseFK</foreign-key-column>
        <key-column>coursePK</key-column>
      </column-map>
    </weblogic-relationship-role>
    <weblogic-relationship-role>
      <relationship-role-name>
         TargetGroupLocal-Has-CourseLocal
      </relationship-role-name>
      <column-map>
        <foreign-key-column>targetGroupFK</foreign-key-column>
        <key-column>targetGroupPK</key-column>
      </column-map>
    </weblogic-relationship-role>
  </weblogic-rdbms-relation>

</ weblogic-rdbms-jar >
```

© Kölsch & Altmann GmbH, 2005

Deployment auf verschiedenen Application Servern

An dieser Stelle sollen keine umfassenden Vergleiche verschiedener Application Server angestellt werden. Sie sollen lediglich grundsätzlich gegenübergestellt werden mit einigen ihrer jeweils wichtigsten Vorteile und Nachteile.

Referenz-Implementierung (SUN), Version 1.3.1

Wie aus dem Namen schon ersichtlich, handelt es sich hierbei nur um eine Referenz-Implementierung, die für den produktiven Einsatz nicht gedacht ist. Hierfür bietet Sun den iPlanet Application Server bzw. – künftig – auch den unter dem Stichwort Sun ONE integrierten Application Server an.

Die Referenzimplementierung stellt also keine Ansprüche auf die optimale Umsetzung an sich elementarer Aspekte wie Performanz, Skalierbarkeit oder auch – was man von einer Referenz erwarten würde – Vollständigkeit. So waren die mit der EJB-Version 2.0 eingeführten Container-Management-Relationships mit sehr heißer Nadel gestrickt, so dass weder Rückwärtsreferenzen noch Selbstreferenzen umsetzbar sind.

Eine weitere Schwäche der Referenzimplementierung ist ihre mangelhafte Konfigurierbarkeit bezüglich realer Einsatzbedingungen, wie zum Beispiel das Aufsetzen auf eine bereits existierende Datenbank.

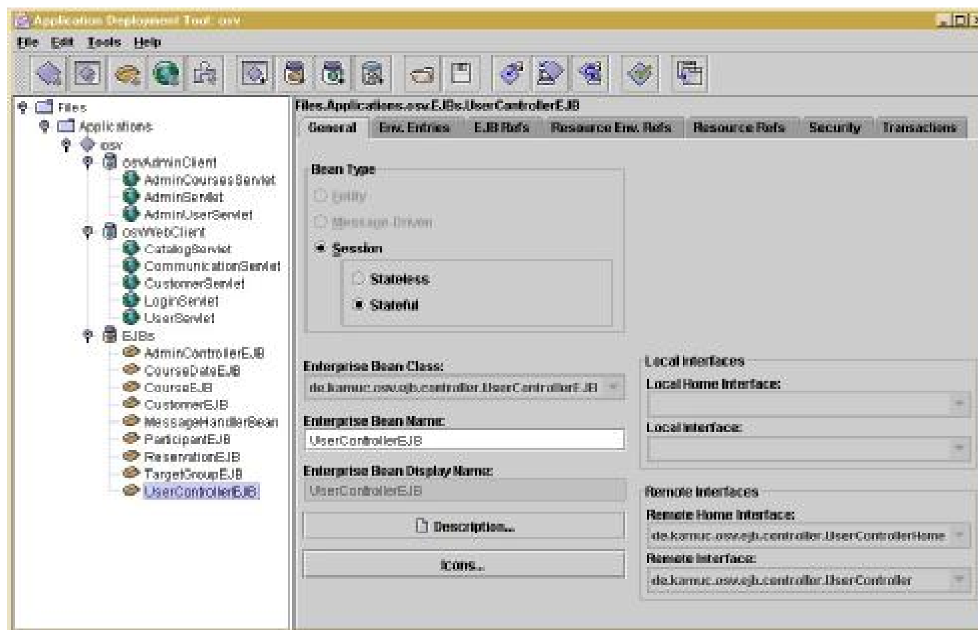
Ihr größter Vorteil ist das relativ leichte Deployment, da man sich bequem durch einige Swing-Formulare klicken kann, um EJB-Archive zu erzeugen. Leider erhält man hierbei kein Gefühl für die mit dem Tool verwalteten XML-Deskriptor-Dateien, was sich beim Umstieg auf andere Server als nicht zu unterschätzendes Problem erweist, da man auf ein entsprechendes Tool meist verzichten muss. Zudem verwischt das Deploy-Tool die Rollen im J2EE Entwicklungsprozess (Developer, Assembler, Deployer), bei kommerziellen Produkten muss diese Trennung jedoch explizit gemacht werden.

Trotz all dieser und weiterer Probleme eignet sich dieser Application Server gut für den Einstieg in die Thematik.

© Kölsch & Altmann GmbH, 2005

K&A

Im Folgenden das Swing-Deploytool bei der Spezifikation einer Session Bean:



Evaluierungslizenz BEA WebLogicServer, Version V6.1

WebLogic wurde von den Javamagazin-Lesern 2002 unter den Application-Servern auf Platz 1 gewählt, dementsprechend hoch sind auch die Erwartungen an diesen Application Server. Die Konfiguration kann hier über einen HTML-Client oder auch über entsprechende Konsolenaufrufe stattfinden. Nach einigem Arbeiten mit dem HTML-Client wird man auch ganz schnell davon ablassen, ist diese doch eher unübersichtlich und man kann sich auch nie sicher sein, ob angeforderte Änderungen an irgendwelchen Deskriptoren auch tatsächlich ausgeführt wurden, auch wenn sie entsprechend quittiert worden sind.

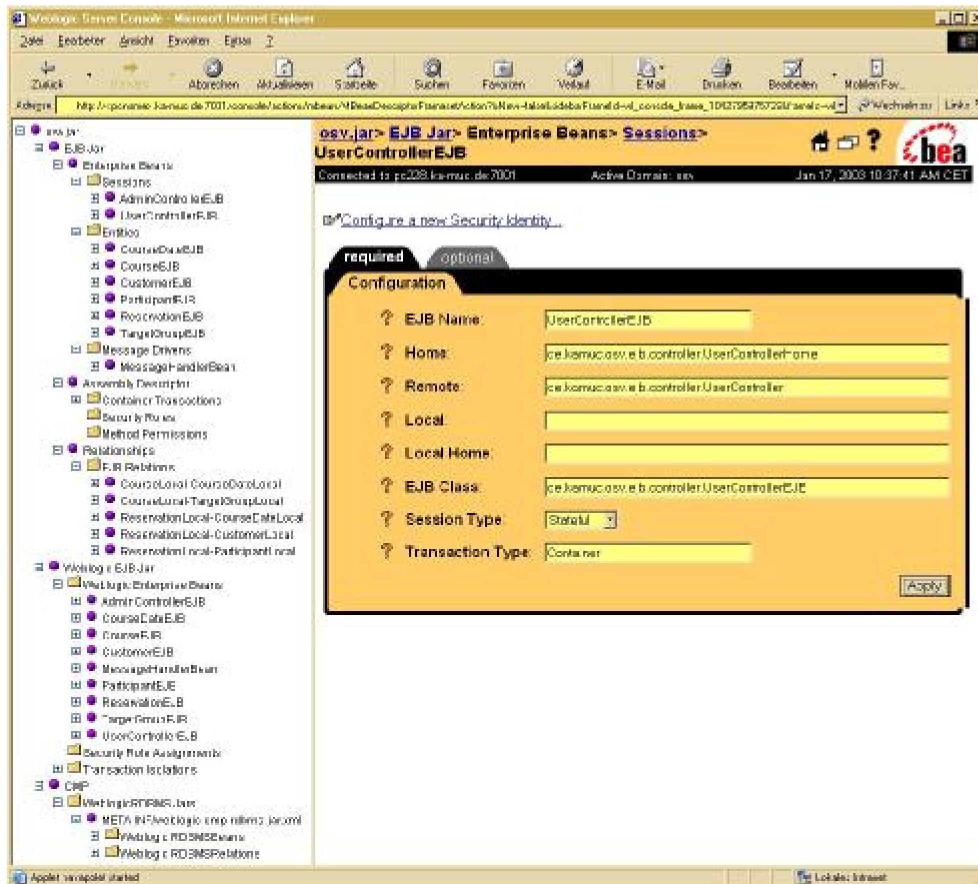
Zudem kann nicht gleichzeitig mit dem HTML-Client und mit Konsolenaufrufen gearbeitet werden. Um nach einer Änderung mit dem HTML-Client wieder mit Konsolenaufrufen arbeiten zu können, muss hierzu immer eine Datei gelöscht werden.

Ist aber der Server erst einmal in Betrieb und korrekt konfiguriert, dann verrichtet er anstandslos seinen Dienst. Vor allem, da die Konsole fast nur zum Einrichten der Ressourcen gebraucht wird. Das Deployment kann dann mittels Kommandozeilenaufrufen (im Lieferumfag enthalten), erledigt werden, die von Ant aus gesteuert werden.

Nicht zuletzt besticht der BEA Weblogic Server durch eine ausführliche und eingängige Dokumentation.

© Kölsch & Altmann GmbH, 2005

Im Folgenden das HTML-Deploytool bei der Spezifikation einer Session Bean:



JBoss V3.0

JBoss ist eine Implementierung der Open-Source-Community des J2EE Standards zur Schaffung eines freien, unabhängigen Application-Servers, der sich immer größerer Beliebtheit erfreut. JBoss wird in verschiedenen Varianten zum Download angeboten, entweder 'standalone' oder mit einem Web-Server gebündelt (Tomcat oder Jetty).

Die Konfiguration ist etwas gewöhnungsbedürftig, da ein relativ einfaches Interface wie bei Sun oder Bea fehlt. Einzig ein spartanisches Web-Interface auf Basis von JMX (Java Management Extension) wird zur Verfügung gestellt. Die restlichen Konfigurationsparameter sind deshalb alle in den entsprechenden Konfigurations-Dateien manuell einzutragen, die dank eingängiger Benennung intuitiv zu finden sind.

Das Deployment stellt sich als äußerst komfortabel dar, da nur in einem Konfigurations-File sog 'Auto-Deployment'-Verzeichnisse anzugeben waren; damit müssen lediglich die EJB-Archive in die entsprechenden Verzeichnisse kopiert werden, woraufhin sie dann automatisch *deployed* werden (analog: *Undeploy* durch löschen).

© Kölsch & Altmann GmbH, 2005

Im Folgenden ein Auszug auf einer Kommandozeilenkonsole:

Leider genehmigt sich JBoss in der untersuchten Version bei den anonymen Primary Keys eine gravierende Abweichung von der Spezifikation, die seinen Einsatz evtl. in Frage stellen könnte.

Zudem hinkt die Dokumentation der Entwicklung zum Teil etwas hinterher.

© Kölsch & Altmann GmbH, 2005

Transaktionen & Isolation-Level

Wir wollen hier nicht beschreiben, was Transaktionen im Einzelnen sind. Hierzu sei auf entsprechende Literatur verwiesen. Herausarbeiten wollen wir nur die i. Zshg. mit EJB relevanten Aspekte. EJB unterstützt verteilte Transaktionen und in diesem Zshg. auch den 2-Phase-Commit. Wir untersuchten, wie die grundlegenden ACID-Anforderungen bei der DB-Programmierung vom EJB-Container umgesetzt werden, bzw. welche Einstellungen hierfür notwendig sind. Einige Aspekte wollen wir im Folgenden erläutern und unsere Erfahrungen v. a. mit dem BEA-WLS (V6.1) weitergeben.

Locking Services

Bei den Locking Services gibt es im Zshg. mit EJB-Containern drei relevante Einstellungen:

- **Exclusive**
  Diese Strategie der Regelung konkurrierender Zugriffe ist die Voreinstellung bei BEA-WLS bis einschließlich den 5.xx-Versionen. Natürlich hat es Vorteile, wenn die Zugriffe auf einzelne Entity Beans serialisiert werden, die Konsistenz der DB-Inhalte ist so zu keinem Zeitpunkt in Gefahr. Ebenso konnte die Anzahl DB-Zugriffe klein gehalten werden.
  Nun ist EJB jedoch konzipiert worden, in hohem Maße zu skalieren. Sind in einer Anwendung konkurrierende Zugriffe auf dieselben Daten häufig, so wird diese Einstellung zum Flaschenhals. Des Weiteren sind bei dieser Einstellung bei konkurrierendem Zugriff sogar Clients blockiert, die Daten nur lesen wollen.
  Zudem muss sich zumindest ein guter Container um Dinge wie z. B. Deadlocks kümmern, Probleme, die auf DB-Ebene längst gelöst sind. Dies ist wohl auch der Grund dafür, dass ab der WLS-Version 6.1 der Default auf die folgende Einstellung geändert wurde:

- **Database**
  Bei dieser Einstellung soll sich die Datenbank um dieses Thema kümmern, nicht jedoch ohne mit weiteren Einstellungen in den Deployment-Deskriptoren die Granularität der zu sperrenden Daten – über Isolation-Levels – feiner einstellen zu können, als dies mit der exklusiven Strategie für konkurrierenden Zugriff möglich ist. Die verschiedenen Isolation-Level werden wir im Folgenden beschreiben.
  Auch bei dieser Locking-Strategie wird sich der Container EJB-Instanzen im Cache vorhalten, dies allerdings nur während Transaktionen. Beim ersten Zugriff innerhalb einer Transaktion werden die Daten eingelesen und am Ende (Commit, Rollback) in die DB geschrieben.
  Diese Strategie lässt auch den Zugriff weiterer Clients (außerhalb des Containers) auf die DB zu, da die Sperrhoheit auf Seiten der DB liegt.

© Kölsch & Altmann GmbH, 2005

K&A

- **ReadOnly**
  Diese Einstellung kann für nur lesende Zugriffe auf bestimmte Daten interessant sein. Hiermit ist sichergestellt, dass auf DB-Ebene keine Sperren gesetzt bzw. bis zum Ende der Transaktionen gehalten werden.

Eingestellt werden können die Locking Services z. B. beim BEA-WLS pro Entity-Bean über das XML-Tag <concurrency-strategy> in der Datei „weblogic-ejb-jar.xml". Bei anderen Application Servern sollte dies ähnlich sein.

Isolation Level

Wir möchten an dieser Stelle nun auf die für den Locking Service „Database" relevanten Isolation Level (siehe hierzu auch [6]) im Zshg. mit EJB eingehen. Der Vollständigkeit halber führen wir an dieser Stelle die durch die Einstellung von Isolation Level gelösten Probleme in aller Kürze noch einmal auf:

- **dirty read**
  Falls eine Transaktion A Informationen in der DB ändert, diese innerhalb einer weiteren Transaktion B nun gelesen werden, so bestehen zwei Möglichkeiten zur Inkonsistenz der innerhalb der Transaktion B gelesenen Daten: Falls die Transaktion A zurückgerollt wird oder die Daten innerhalb der Transaktion A wiederholt geändert und in die DB zurück geschrieben werden.

- **non-repeatable read**
  Falls innerhalb einer Transaktion A Daten gelesen werden, die innerhalb einer weiteren Transaktion B geändert und in die DB zurück geschrieben werden, so ist es innerhalb der Transaktion A nicht möglich, bei einer weiteren Leseoperation dieselben Daten zu erhalten wie beim vorigen Versuch.

- **phantom read**
  Werden innerhalb einer Transaktion A im Rahmen einer Query eine Reihe Daten gelesen, so ist es innerhalb dieser Transaktion nicht möglich, dasselbe Ergebnis wiederholt zu erhalten, falls innerhalb einer weiteren Transaktion B Daten, die die Bedingungen der Query erfüllen, neu in die DB eingebracht oder aus dieser gelöscht wurden.

Um diesen Problemen zu begegnen, sind auf Ebene der DB entsprechende Isolation Level definiert. So begegnet

- der Isolation Level *ReadUncommitted* keinem der obigen Probleme
- der Isolation Level *ReadCommitted* dem „*dirty read*"-Problem
- der Isolation Level *RepeatableRead* zusätzlich dem „*non-repeatable read*"-Problem
- der Isolation Level *Serializable* zusätzlich dem „*phantom read*"-Problem

Für Oracle z. B. ist gar ein weiterer Isolation Level *ReadCommittedForUpdate* definiert. Bei dieser Einstellung wird sichergestellt, dass schon beim Lesen von Daten für diese Schreibsperren angefordert und gesetzt werden. Allerdings ist eine solche Strategie kontraproduktiv zur Strategie des optimistischen Sperrens, das sich auch bei Application Servern zunehmend durchsetzt. Abhängig ist dies

© Kölsch & Altmann GmbH, 2005

allerdings wiederum auch davon, wie hoch die Wahrscheinlichkeit von konkurrierendem Zugriff ist.

Leider konnten sich die Standardisierungsgremien für die EJB-Spezifikation – wie im Übrigen auch schon bei den Locking Services – nicht darauf einigen, diese Überlegungen in der Spezifikation Rechnung zu tragen. Insofern bleibt es Container-Herstellern überlassen, inwieweit und in welcher Form Einstellungen möglich sind. Des Weiteren ist zu erforschen, inwieweit eingesetzte DBMS diese Isolation Level auch unterstützen (Oracle unterstützt z. B. neben dem nicht standardisierten Isolation Level *ReadCommittedForUpdate* nur noch die Isolation Level *ReadCommitted* (Default) und *Serializable*).

Beim BEA-WLS sind in der Datei „*weblogic-ejb-jar.xml*" jedem Isolation Level die Methoden zuzuordnen, mit der DB-Zugriffe innerhalb dieser gefahren werden sollen. Nun ist das Einstellen eines Isolation Level allerdings nur als allererstes SQL-Kommando innerhalb einer Transaktion möglich, eine Transaktion kann sich aber über mehrere Methoden erstrecken. Insofern sind die Einstellungen natürlich nur für die Methoden interessant, die auch am Anfang einer Transaktion stehen. Bei CMP sind dies in der Regel die von Clients aufgerufenen Business-Methoden, falls sie den Transaktionskontext nicht schon vom Client übergeben bekommen und diesen lediglich fortführen (siehe hierzu auch XML-Tag „*trans-attribute*").

Im folgenden ein Beispiel für eine solche Einstellung im Deployment-Deskriptor:

```
<transaction-isolation>
  <isolation-level>TRANSACTION_SERIALIZABLE</isolation-level>
    <method>
      <ejb-name>UserControllerEJB</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>reserveCourse</method-name>
      <method-params>
         <method-param>
            de.kamuc.osv.ejb.customer.CustomerData
         </method-param>
         <method-param>
            de.kamuc.osv.ejb.reservation.ReservationData
         </method-param>
      </method-params>
    </method>

    <!-- weitere Methoden mit diesem Isolation-Level-->

</transaction-isolation>

<!-- weitere Transaction-Isolation-Level mit zugeordneten
     Methoden -->
```

Verwendung von Ressourcen

Grundsätzlich sind Resource-Manager-Connection-Factories vom Bean-Entwickler nur zu referenzieren bzw. deklarieren und erst vom Bean-Deployer zu definieren. Im Wesentlichen gibt es hier JDBC-Connection-Factories und JMS-Resource-

© Kölsch & Altmann GmbH, 2005

Manager-Connection-Factories. Eine solche Deklaration könnte beispielsweise folgendermaßen aussehen

```
<resource-ref>
   <description>...</description>
   <res-ref-name>jdbc/OracleDB</res-ref-app>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
   ...
</resource-ref>
```

und findet auf Bean-Ebene statt. Bei CMP ist eine solche Einstellung – zumindest beim BEA-WLS – bzgl. Datenquellen nicht notwendig, da das Mapping über JNDI-Namen stattfindet, die im Container-spezifischen Deployment-Deskriptor („*weblogic-cmp-rdbms-jar.xml*") festgelegt werden. Darin wird für jede Entity Bean das Mapping auf die DB festgelegt und unter anderem auch der JNDI-Name festgelegt, über den dann auf die beim entsprechenden Server festzulegenden Ressource-Manager zugegriffen werden kann. Mit diesen – deklarativen – Einstellungen kann bei CMP der Container die Anbindung an die DB übernehmen.

Mit Deklarationen wie in obigem Beispiel erschöpft sich denn auch schon der Standard, weiteres ist innerhalb der Container-spezifischen Deployment-Deskriptoren festzulegen. Insofern beziehen sich die weiteren Ausführungen hierzu hauptsächlich auf den BEA-WLS:

Ressource: JDBC-Datenquellen

In der bzgl. WLS und Oracle spezifischen Deployment-Deskriptor-Datei („*weblogic-cmp-rdbms-jar.xml*") ist für jedes Entity-Bean ein JNDI-Name für die JDBC-Datenquelle (<data-source-name>) zu definieren, über die der Zugriff zur DB erfolgen soll. Diesen JNDI-Namen müssen nun real eingerichtete JDBC-Datenquellen zugeordnet werden. Konfiguriert werden diese beim WLS-Server, also in der entsprechenden Konfigurationsdatei („*config.xml*").

Zunächst einmal gibt es zwei Arten von Transaktionen, lokale und verteilte. EJB-Container arbeiten grundsätzlich mit verteilten Transaktionen, d.h. eine Transaktion kann mehrere Datenbanken mit einbeziehen. Interessant wird in diesem Zusammenhang auch der sogenannte 2-Phase-Commit, der Teil des XA-Standards ist. In diesem Sinne gibt es auch zwei verschiedene Arten von Datenquellen:

- **JDBCDataSource**:
  Bietet Connection-Pooling-Service auf der Grundlage von purer JDBC-Funktionalität an.

- **JDBCTxDataSource**:
  Bietet zusätzlich noch den Service der Teilnahme an JTS (Java Transaction Service) –Transaktionen an.

Da EJB – insbesondere im Zshg. mit CMP auf JTS aufsetzt, ist eine solche JDBCTxDataSource grundsätzlich zu verwenden, auch dann, wenn nur eine

© Kölsch & Altmann GmbH, 2005

Datenbank beteiligt ist. Das Problem mit einer JDBCDataSource i. Zshg. mit EJB ist nämlich, dass nach jedem einzelnen DB-Aufruf implizit sofort ein Commit ausgeführt wird („autocommit=true“), was für die Verwendung von Transaktionen natürlich nicht sehr förderlich ist. Insbesondere ist es dann auch nicht möglich, im Fehlerfalle schon abgesetzte DB-Kommandos wieder rückgängig zu machen.

Generell ist eine TxDataSource zu verwenden, wenn

- die Java Transaction API (JTA) verwendet wird,
- ein Application Server mit einem EJB-Container verwendet wird,
- mehrere Datenbanken in eine Transaktion einbezogen werden sollen,
- wenn innerhalb einer Transaktion auf mehrere Resourcen, wie z. B. Datenbank und Java Message Service (JMS), zugegriffen wird,
- wenn ein und derselbe Connection-Pool auf mehreren Servern verwendet wird.

Ein entsprechender Eintrag in die Konfigurationsdatei des Application Servers muss folgender Form entsprechen:

```
<JDBCTxDataSource
  JNDIName="<jndiName>"
  Name="OracleDataSource"
  PoolName="OracleDB"
  Targets="<serverName>" />
```

Hierbei kann die Datenquelle über den JNDI-Namen von EJBs angesprochen und genutzt werden. Der Name der Datenquelle ist frei wählbar, der Pool-Name verweist auf einen ebenfalls zu definierenden Connection-Pool.

Connection-Pools

Vorgaben bzgl. Connection-Pooling sind nicht Bestandteil der EJB-Spezifikation, werden aber von den bedeutenden Container-Herstellern angeboten. Spezifiziert wird lediglich, dass es für die Beans transparent sein muss, ob Pooling stattfindet oder nicht.

Es werden immer wieder Verbindungen zu einzelnen Datenbanken benötigt. Da diese Ressourcen nicht unendlich sind, müssten sie nach Gebrauch auf jeden Fall sofort wieder freigegeben werden. Da das Anlegen von Ressourcen zudem aber auch noch teuer ist, sich der EJB-Entwickler zudem nicht damit herumschlagen möchte, ist die Idee, eine Menge von DB-Verbindungen in einem Pool zu halten. Eine minimale Anzahl von DB-Verbindungen kann vom Application Server schon beim Hochfahren angelegt werden, Anforderungen können zunächst aus diesem Pool befriedigt werden, weitere können bei Bedarf bis zu einer maximalen Anzahl angelegt werden, nicht mehr benötigte DB-Verbindungen werden wieder in den Pool zurückgestellt und können in verkehrsschwachen Zeiten auch wieder freigegeben werden.

Jeder Datenquelle muss nun ein entsprechender Connection-Pool zugeordnet werden. Dieser definiert insbesondere den entsprechenden JDBC-Treiber und die URL, wo die DB erreicht werden kann. Im folgenden ein entsprechender Eintrag:

© Kölsch & Altmann GmbH, 2005

```
<JDBCConnectionPool
  DriverName="oracle.jdbc.driver.OracleDriver"
  Name="OracleDB"
  Password="<crypted representation>"
  Properties="user=<user>"
  Targets="<serverName>"
  URL="jdbc:oracle:thin:@<host(ipAddressOrName)>:1526:
       <dbInstanceName>" />
```

Die Größe des Pools, also die minimale und maximale Anzahl, das Inkrement beim Wachsen und das Verhalten beim Freigeben von DB-Verbindungen kann über weitere Parameter gesteuert werden.

Eine Besonderheit ergibt sich in diesem Beispiel mit dem angegebenen Treiber, der kein XA-Treiber ist. Es ist erlaubt, dass für höchstens eine Datenquelle ein non-XA-Treiber verwendet wird. An einer Transaktion darf allerdings höchstens ein non-XA-Treiber beteiligt sein. Dies ist insbesondere nützlich bei Verwendung älterer DBMS und DBMS-Versionen, die den XA-Standard noch nicht unterstützen. Ist ein non-XA-Treiber an verteilten Transaktionen beteiligt, so muss für dessen Datenquelle allerdings das Property „*EnableTwoPhaseCommit*" auf „*true*" gesetzt werden.

Interessant ist in diesem Zshg. auch, dass beim BEA-WLS die Oracle-Treiber z. B. schon mit installiert sind, sollen andere verwendet werden, müssen sie in der Start-Datei des WLS mit in den dort definierten CLASSPATH eingetragen werden, bei Verwendung des gleichen Treibers (z. B. in einer anderen Version) ist ein solcher im CLASSPATH natürlich vor den WLS-spezifischen JAR-Dateien einzutragen.

© Kölsch & Altmann GmbH, 2005

JNDI & EJB-Referenzen

Der Sinn des JNDI-Lookup-Mechanismus besteht für EJB-Anwendungen darin, vom Source-Code aus auf externe Ressourcen zugreifen zu können, ohne dass bei der Erstellung des Source-Codes die später verwendeten konkreten Ressourcen bekannt sein müssen und ohne dass bei Bereitstellung dieser Ressourcen der Source-Code angepasst werden muss.

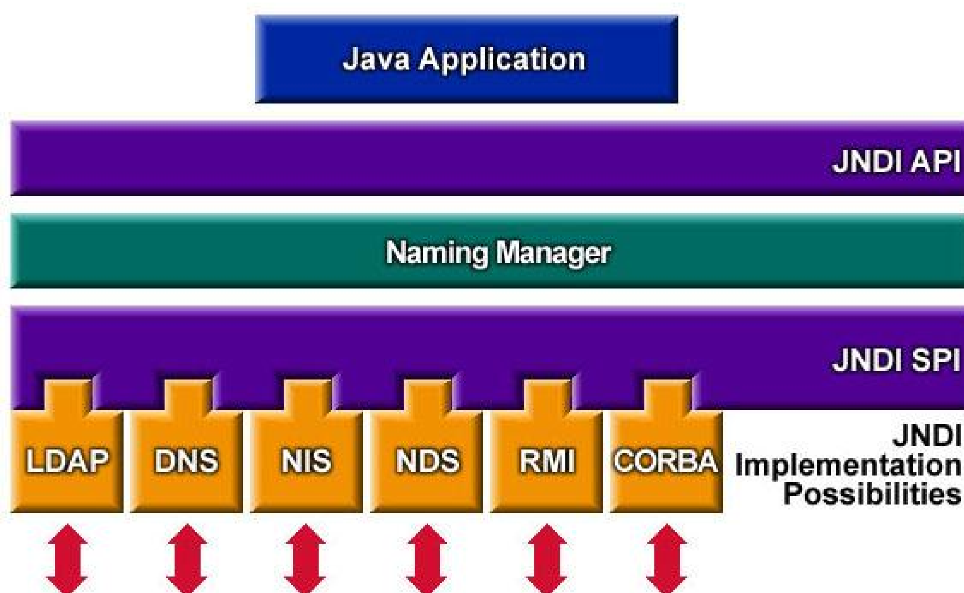


Abbildung 1: JNDI-Architektur (Quelle: SUN-Tutorial bzgl. JNDI)

Die Abbildung der im Source-Code referenzierten symbolischen Ressource-Namen auf die tatsächlichen Ressourcen einer konkreten Ablaufumgebung erfolgt zum Zeitpunkt des Deployments durch entsprechende Eintragungen in den Deskriptoren. Zu solchen Ressourcen gehören neben Datenbankverbindungen, JMS-Connections und Umgebungsvariablen auch vom Source-Code referenzierte EJB's.

Die Ressourcen sind dem Source-Code über logische Namen zugänglich. Damit diese Namen vom Source-Code-Entwickler frei gewählt werden können und nicht mit Namen anderer Beans kollidieren (und somit zum Deployment-Zeitpunkt nicht geändert werden müssen), wird pro Komponente, die ein Lookup durchführt, ein eigener Namensraum aufgespannt. Dazu muss beim Lookup-Aufruf der logische Name zwingend mit "java:comp/env/<logischer Name>" angesprochen werden. Obwohl es möglich ist, beim Lookup ohne "java:comp/env" auszukommen, ist dies nicht ratsam, da damit direkt der Server-weit globale JNDI-Name im Lookup-Aufruf verwendet werden müsste und somit die Grundidee des JNDI-Lookup bei J2EE-Anwendungen, nämlich die Entkopplung des Source-Code von der Ablaufumgebung, umgangen würde.

© Kölsch & Altmann GmbH, 2005

Im weiteren Verlauf des Kapitels wird anhand von EJB-Referenzen an einem Beispiel veranschaulicht, welche Eintragungen in welchen Deskriptoren nötig sind, damit eine korrekte Anwendung des JNDI-Lookup-Mechanismus stattfindet. Als J2EE-Server-spezifische Deskriptoren werden die aus der Arbeit mit dem BEA WLS entwickelten Deskriptoren herangezogen.

Mit EJB-Referenzen gibt der Bean Provider an, welche Beans von seiner Implementierung referenziert werden, oder genauer gesagt, wie Home- und Component-Interface der referenzierten Bean heißen. Wie für andere externe Ressourcen auch, wird für die Auflösung von EJB-Referenzen der JNDI-Lookup-Mechanismus verwendet. Dabei wird der im Source-Code verwendete logische Name (auch "coded name" genannt) in einem J2EE-Server-spezifischen Deskriptor auf einen JNDI-Namen abgebildet. Unter diesem JNDI-Namen ist die zu referenzierende Bean beim JNDI-Namensdienst registriert. Dieses Registrieren geschieht zum Deployment-Zeitpunkt durch den J2EE-Server, der aus einem Server-spezifischen Deskriptor den JNDI-Namen der zu registrierenden Bean liest.

Die Auflösung von Referenzen auf externe Ressourcen funktioniert grundsätzlich für alle Arten von Ressourcen nach dem gleichen Schema. Für die Auflösung von EJB-Referenzen gibt es jedoch ein (optionales) Hilfsmittel, den sog. "ejb-link", welcher in dem nachfolgenden Beispiel verwendet wird. Zu betonen ist, dass sich durch die Verwendung des ejb-link nichts am JNDI-Lookup-Mechanismus ändert. Er stellt lediglich eine Arbeitserleichterung für die EJB-Rollen Application Assembler und Deployer dar.

Der ejb-link wird folgendermaßen eingesetzt:

Bei der Zusammenstellung der Applikation kann der Application Assembler im Deskriptor ejb-jar.xml mit Hilfe des <ejb-link>-tags die EJB-Referenzen auflösen. Zum Deployment-Zeitpunkt werden die *"ejb-link"*s vom Server in JNDI-Namen (in den Server-spezifischen Deskriptoren) umgewandelt, also in die Form, in der sie der Deployer hätte eingeben müssen, wenn keine ejb-links verwendet worden wären.

Zu beachten ist, dass ejb-link nur innerhalb einer Applikation anwendbar ist.

Die Vorteile der Verwendung von "*ejb-link*"s sind:

- Der Application Assembler dokumentiert damit gegenüber dem Deployer, welche EJB-Referenzen er wie (also mit welchen konkreten Bean-Implementierungen) aufgelöst hat und schließt so den Assembly-Vorgang ab.
- Der Deployer muss lediglich den JNDI-Namen angeben, unter dem die referenzierte Bean beim JNDI-Namensdienst registriert wird. Die Komponenten, die die Bean referenzieren, benutzen den *"ejb-link"*, wodurch das u. U. häufige Vorkommen von Mappings zwischen "*coded name*" und JNDI-Name in den Deskriptoren wegfällt. Dies wird, wie erwähnt, vom Server zum Deployment-Zeitpunkt übernommen und erleichtert so die Arbeit des Deployers, der durch passende Namenswahl des öfteren für eindeutige JNDI-Namen sorgen muss und dann nur noch an einer Stelle den JNDI-Namen anpassen muss.

Im Folgenden werden die für den JNDI-Lookup einer EJB-Referenz notwendigen Eintragungen in den Deskriptoren unter Verwendung von "*ejb-link*" aufgeführt. Die

© Kölsch & Altmann GmbH, 2005

anfallenden Arbeitsschritte sind nach den Rollen Bean Provider, Application Assembler und Deployer gegliedert.

Der Bean Provider liefert Bean-Implementierungen zusammen mit dem Deskriptor "*ejb-jar.xml*". In diesem Deskriptor sind alle in dem jar-File enthaltenen Beans beschrieben. Außerdem sind mittels des tags "*<ejb-local-ref>*" (bzw. ggf. "*<ejb-ref>*") die Referenzen auf andere Beans beschrieben, oder präziser, die Interfaces, die eine Bean erfüllen muss, um die vom Bean Provider geforderte Referenz aufzulösen.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>AdminControllerEJB</ejb-name>
      <home>
        de.kamuc.osv.ejb.controller.AdminControllerHome
      </home>
      <remote>
        de.kamuc.osv.ejb.controller.AdminController
      </remote>
      <ejb-class>
        de.kamuc.osv.ejb.controller.AdminControllerEJB
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-local-ref>
        <ejb-ref-name>ejb/codedNameCourse</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>de.kamuc.osv.ejb.CourseLocalHome</local-home>
        <local>de.kamuc.osv.ejb.CourseLocal</local>
      </ejb-local-ref>
    </session>

   ...

</ejb-jar>
```

Der Source-Code des Bean Providers, in dem das Lookup durchgeführt wird, sieht folgendermaßen aus:

```
ctx = new InitialContext ();
courseHome = (CourseLocalHome)
               ctx.lookup("java:comp/env/ejb/codedNameCourse");
```

Der Application Assembler beschafft alle Komponenten und fügt sie zu einer Applikation zusammen. Dabei wählt er die Beans aus, die die vom Bean Provider in den EJB-Referenzen geforderten Interfaces erfüllen, ordnet die Beans unter Verwendung von <ejb-link>-tags zu, und löst somit die EJB-Referenzen auf.

© Kölsch & Altmann GmbH, 2005

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>AdminControllerEJB</ejb-name>
      <home>
        de.kamuc.osv.ejb.controller.AdminControllerHome
      </home>
      <remote>
        de.kamuc.osv.ejb.controller.AdminController
      </remote>
      <ejb-class>
        de.kamuc.osv.ejb.controller.AdminControllerEJB
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-local-ref>
        <ejb-ref-name>ejb/codedNameCourse</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>de.kamuc.osv.ejb.CourseLocalHome</local-home>
        <local>de.kamuc.osv.ejb.CourseLocal</local>
        <ejb-link>CourseEJB</ejb-link>
      </ejb-local-ref>
    </session>

     ...

  </enterprise-beans>
</ejb-jar>
```

Das Ergebnis ist ein Konglomerat von Komponenten, deren Referenzen auf konkrete Implementierungen weisen. Es stellt die "statische" Sicht der zusammengestellten Applikation dar.

Die Aufgabe des Deployers ist die Anbindung externer Ressourcen und die Vergabe von beliebigen, jedoch Server-weit eindeutigen JNDI-Namen für diese Ressourcen. Da diese Deployment-Einstellungen vom verwendeten J2EE-Server abhängen, werden diese in Server-spezifischen Deskriptoren vorgenommen. In Falle des BEA Weblogic Servers ist es folgender Deskriptor ("*weblogic-ejb-jar.xml*"):

```
<weblogic-ejb-jar>
  <weblogic-enterprise-beans>
    <ejb-name>CourseEJB</ejb-name>

    ...

    <local-jndi-name>JNDIName1</local-jndi-name>
  </weblogic-enterprise-beans>

  ...

</weblogic-ejb-jar>
```

© Kölsch & Altmann GmbH, 2005

K&A

Java Message Service im J2EE-Kontext

Im Folgenden soll kurz das Konzept der nachrichtenorientierten Kommunikation mit Hilfe von *Messaging Services* vorgestellt, und im Anschluss deren Einsatzmöglichkeiten bei der Entwicklung im Umfeld der J2EE-Plattform aufgezeigt werden.

Messaging Services

Mit Hilfe eines Messaging Service lässt sich eine ansynchrone Kommunikation zwischen Komponenten eines Softwaresystems umsetzen. Man spricht in diesem Fall von einer *losen Kopplung* der Komponenten. Die besonderen Eigenschaften dieser losen Kopplung sind dabei:

- Asynchrone Kommunikation, also die zeitliche Entkopplung zwischen dem Senden einer Nachricht und deren Empfangen bzw. Bearbeitung in der empfangenden Komponente. Und auch zwischen dem evtl. zu erwartenden Ergebnis auf Grund einer gesendeten Nachricht.

- Die Funktionsschnittstellen der jeweils anderen Komponente (Interfaces) müssen nicht bekannt sein. Die Kommunikation geschieht in diesem Fall über den spezifizierten Aufbau der übermittelten Nachrichten.

Bereitgestellt wird diese Funktionalität durch eine so genannte *Message Oriented Middleware* (MOM), die für die Entgegennahme von Nachrichten von einem Sender und deren Weiterleitung an einen oder mehrere Empfänger verantwortlich ist. Hierzu existieren zwei verschiedene Ansätze, die in **Abbildung 1** und **Abbildung 2** dargestellt sind. Welcher der beide Ansätze verwendet wird, ergibt sich in der Regel aus dem Kontext der zu entwickelnden Anwendung und deren konkreten Anforderungen.

Point-to-Point

Beim *Point-to-Point* Ansatz verwaltet die MOM eingegangene Nachrichten eines Senders in einer *Queue*, aus der diese entnommen und an einen Empfänger weitergereicht werden. Eine Nachricht wird in diesem Fall also jeweils nur von *einem* Empfänger bearbeitet.

© Kölsch & Altmann GmbH, 2005

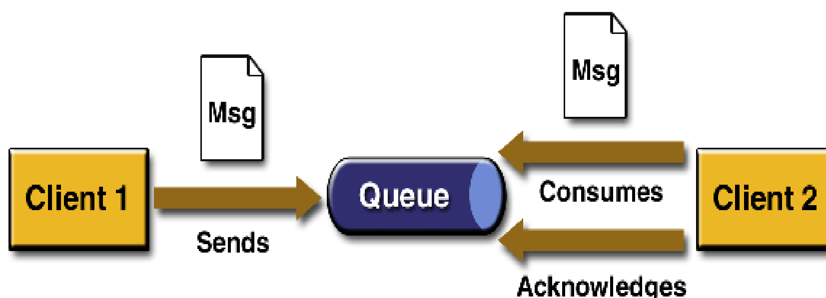


Abbildung 1: *Point-to-Point* Ansatz für den Nachrichtenaustausch (Quelle: Sun Microsystems)

Publish/Subscribe

Beim *Publish/Subscribe* Ansatz kann eine Nachricht von mehreren Empfängern verarbeitet werden. Dazu werden von der MOM s.g. *Topics* verwaltet, die von Empfängern abonniert werden können. Jeder Empfänger, der ein bestimmtes Topic abonniert hat, bekommt die Nachrichten die unter diesem Topic veröffentlicht werden zugestellt.

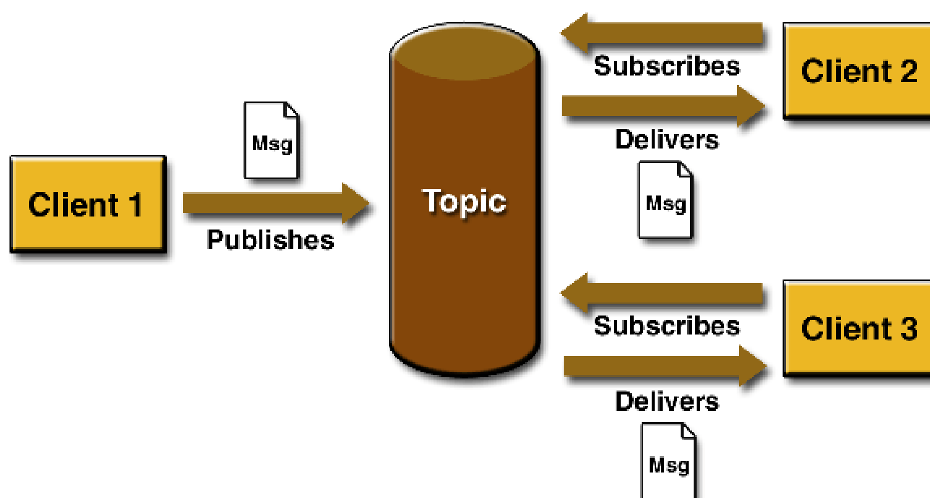


Abbildung 2: *Publish/Subscribe* Ansatz (Quelle: Sun Microsystems)

Etwas abstrakter betrachtet besteht in einer Anwendung der Teil für die nachrichtenorientierte Kommunikation aus den Sendern und Empfängern einer Nachricht (auch als *Message-Producer* und *Message-Consumer* bezeichnet), sowie aus der bereits erwähnten MOM, also einem Messaging-Server, der die Verwaltung und Übermittlung der Nachrichten realisiert. Dazu stellt er den Sendern und Empfängern die für sie notwendigen Dienste zur Verfügung, um Nachrichten senden und empfangen zu können.

© Kölsch & Altmann GmbH, 2005

Java Message Service

Für die Java-Plattform wurde die *Java Message Service* API (JMS-API) spezifiziert. Mit diesem kann — in Verbindung mit einem so genannten *JMS-Provider* (Messaging-Server) — die nachrichtenorientierte Kommunikation in Java-Anwendungen implementiert werden. Stellt ein Hersteller eines Messaging-Systems eine Implementierung des JMS-API in Form eines *JMS-Providers* für sein Produkt zur Verfügung, kann auf diese Weise theoretisch jedes Messaging-System an eine Java-Anwendung angebunden bzw. neue Java-Anwendungen beispielsweise in bestehende Workflowsysteme integriert werden.

In **Abbildung 3** sind die Komponenten einer JMS-Anwendung dargestellt. Eine besondere Gruppe bilden darin die administrierbaren Komponenten. Zu diesen zählen die *Connection Factories* und die *Destinations*.

- Bei den *Destinations* handelt es sich entweder um ein *Topic* oder eine *Queue* (siehe Abschnitte *Point-to-Point* und *Publish/Subscribe*).
- Über eine *Connection Factory* erhalten die Sender und Empfänger eine Verbindung zum *JMS-Provider*, über die wiederum eine Session zur Nachrichtenübermittlung eröffnet werden kann.

Eine Session bildet nun den notwendigen Kontext einer Nachrichtenübertragung, über den auch die Steuerung von Transaktionen für die Nachrichtenübertragung zur Verfügung steht.

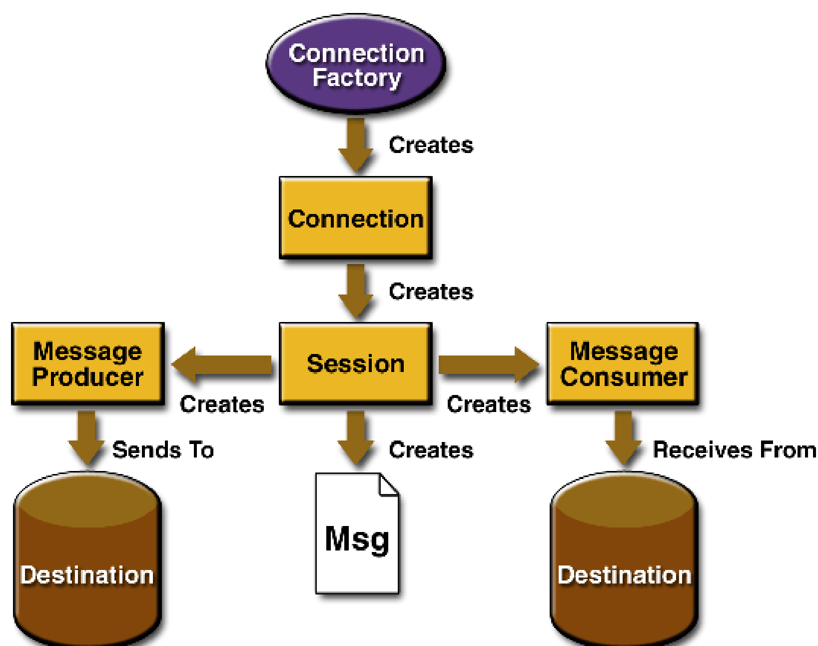


Abbildung 3: Komponenten einer JMS-Anwendung (Quelle: Sun Microsystems)

Wenn es darum geht, robuste und komplexe verteilte Anwendungen zu entwickeln, stellt sich JMS als attraktive Basis für die Anwendungsarchitektur dar. Denn zum

© Kölsch & Altmann GmbH, 2005

einen bietet es die Zusicherung der Zustellung von Nachrichten durch die JMS-API selbst, wodurch die Komplexität der selbst zu entwickelnden Komponenten diesbezüglich gering gehalten wird. Zum anderen verfügt es über weitere Features, wie die Einbeziehung der Nachrichtenübermittlung in verteilten Transaktionen und die so genannten *Durable Subscriptions*.

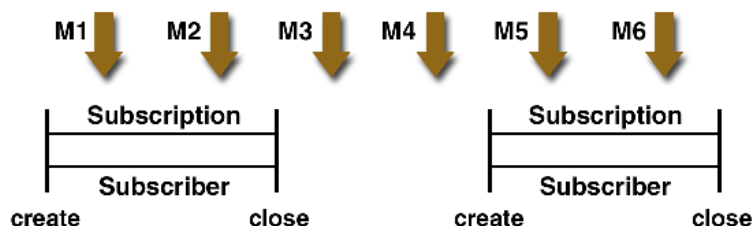


Abbildung 4: M3 und M4 werden vom Subscriber nicht empfangen (Quelle: Sun Microsystems)

Beim Publish/Subscribe Modell ermöglichen die *Durable Subscriptions* die Zustellung von Nachrichten eines Topics auch an Consumer, die nicht durchgehend eine Session zu diesem Topic besitzen. Somit erhalten auch Consumer beispielsweise nach einem Absturz eines Systems alle Nachrichten, die an das Topic versendet wurden während sie nicht aktiv waren. (siehe **Abbildung 4** und **Abbildung 5**)

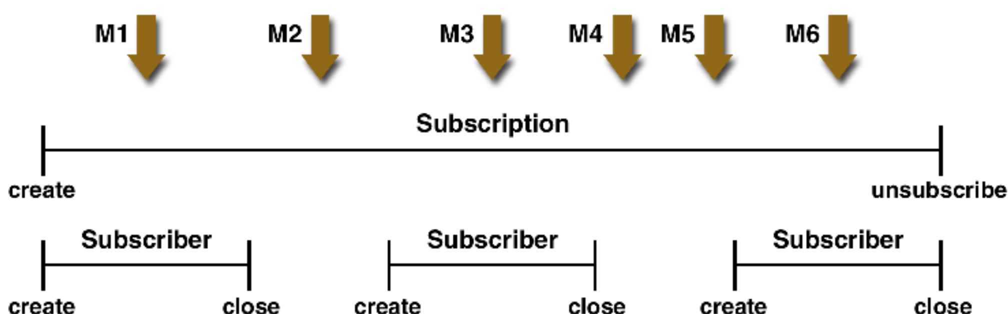


Abbildung 5: Auch M2, M4 und M5 werden vom Subscriber empfangen (Quelle: Sun Mircosystems)

Einsatz von JMS in J2EE-Anwendungen

Das JMS-API ist fester Bestandteil der J2EE-Plattform, und somit zusätzlich zur J2EE-Connector-Architecture (JCA) eine weitere Möglichkeit zur Anbindung von J2EE-Anwendungen an vorhandene Systeme. Aber auch die gängigen Application-Server enthalten einen JMS-Provider, und stellen somit die Infrastruktur für nachrichtenbasierte Komponenten von J2EE-Anwendungen bereit. Zu beachten ist bei der Produktauswahl jedoch die Tatsache, dass nicht bei allen auf dem Markt befindlichen Application-Server oder JMS-Provider alle oben beschriebenen Funktionalitäten implementiert wurden.

© Kölsch & Altmann GmbH, 2005

Message Driven Beans

Unterscheidet sich das Versenden von Nachrichten in J2EE-Anwendungen noch nicht zum Versenden einer Nachricht in einer reinen JMS-Anwendung, so bietet die J2EE-Plattform zum Empfang von Nachrichten eine Erweiterung, die *Message Driven Beans* (MDB). Diese MDBs sind in einer J2EE-Anwendung die Consumer einer Nachricht (siehe **Abbildung 3**), und realisieren ausschliesslich das asynchrone Empfangen von Nachrichten. Synchrones Empfangen von Nachrichten in Session oder Entity Beans ist nicht empfehlenswert.

Ebenfalls unterschiedlich gestaltet sich das Steuern von Transaktionen innerhalb eines MDB im Vergleich zu einem Consumer einer JMS-Anwendung außerhalb der J2EE-Plattform. Die Transaktionen werden hier in der Regel vom EJB-Container bereitgestellt. Diese können, wie bei Session Beans und Entity Beans, vom MDB über dessen EJB-Context beeinflusst werden.

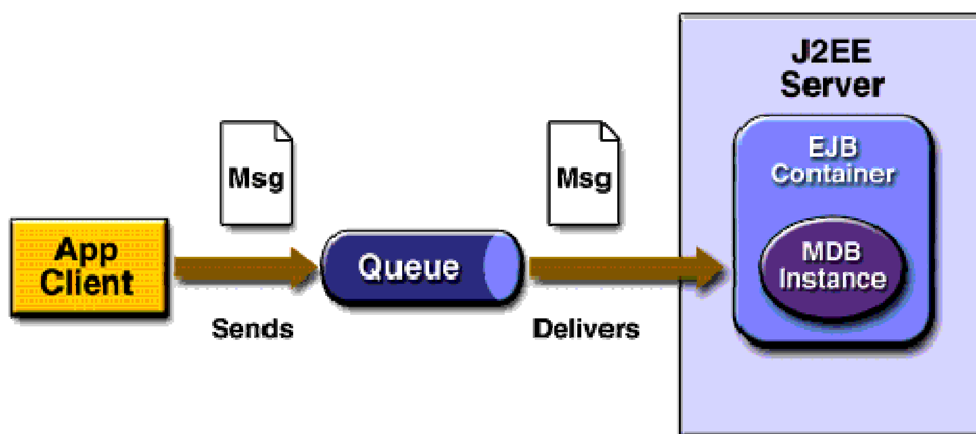


Abbildung 6: *Message Driven Bean* als *Consumer* von Nachrichten in einer J2EE-Anwendung (Quelle: Sun Microsystems)

Genau wie Stateless Session Beans besitzen auch die MDBs keinen Zustand, und sind somit gut geeignet, die Lasten zu verteilen, die durch eingehende Nachrichten in einer J2EE-Anwendung entstehen können. So kann ein Application-Server mehrere Instanzen eines MDB erzeugen, die die eingehenden Nachrichten parallel verarbeiten können.

In **Abbildung 6** ist ein einfaches Szenario für die Verwendung von MDBs in einer J2EE-Anwendung dargestellt. Auf der Seite des Client ergibt sich durch die Verwendung der asynchronen, nachrichtenorientierten Kommunikation, neben den bereits erwähnten Vorteilen auch eine bessere Antwortzeit aus Sicht des Benutzers. In der Regel ist in einem solchen Szenario kein sofortiges Ergebnis auf Grund der versendeten Nachricht notwendig. Der Benutzer muss also nicht die Beendigung eines Aufrufs über ein Remote Interface auf einem Objekt auf dem Application-Server abwarten. Dieser Aufruf würde wesentlich mehr Zeit in Anspruch nehmen.

Einsatzmöglichkeit: Logging mit Message Driven Beans

© Kölsch & Altmann GmbH, 2005

Neben der Integration von J2EE-Anwendungen in existierende Systeme (z.B. Workflowsysteme) oder der Anbindung von existierenden Systemen als Informationsquelle für J2EE-Anwendungen, bietet sich JMS auch für die Kommunikation zwischen den serverseitigen Komponenten innerhalb einer J2EE-Anwendung an.

Da in einer verteilten Anwendung ein lokales Logging ungeeignet erscheint, könnte beispielsweise das Logging der serverseitigen Klassen über ein MDB laufen. D.h. ein Logging-Eintrag wird von den Anwendungsklassen in Form einer JMS-Message , bzw. über das Logging-API, lokal erzeugt und an ein MDB versendet. Dieses MDB ist die Anlaufstelle für alle Logging-Einträge, und bildet so eine Art Zwischenpuffer für die eingehenden Logging-Einträge. Die Pufferung der Logging-Einträge entsteht auf Grund der Tatsache, dass ein Application-Server mehrere Instanzen eines MDB erzeugen kann. Für die Anwendungsklasse, also das Session oder Entity Bean ergibt sich dadurch der bereits angesprochene Laufzeitvorteil, da die Logging-Einträge nicht über eine Methode eines Remote Interface abgesetzt werden müssen, und die Anwendungsklasse somit auch nicht die vollständige Bearbeitung des Logging-Eintrags abwarten muss.

Im MDB kann nun die vergleichsweise mehr Zeit in Anspruch nehmende Behandlung des Logging-Eintrags durchgeführt werden. Mehraufwand ist bei diesem Ansatz dafür bei die Verwendung von Zeitstempeln in einem Logging-Eintrag notwendig. Damit diese weiterhin in der korrekten zeitlichen Reihenfolge ausgewertet werden können, kann nicht die Zeit verwendet werden, zu der der Logging-Eintrag in der Anwendungsklasse ausgelöst wird. Die Anwendungsklassen können sich in diesem Fall auf unterschiedlichen Server befinden, für die nicht zwingend eine gemeinsame Systemzeit verfügbar ist.

Auch die Empfangszeit eines Logging-Eintrags über die JMS-Message im MDB ist nicht brauchbar, da JMS keine zeitlichen Zusicherungen, was das tatsächliche Versenden der JMS-Message und deren Zustellung durch den JMS-Provider angeht, spezifiziert. Nach Überwindung dieser Schwierigkeiten ist das Logging über JMS in verteilten Anwendungen aber ein guter Ansatz.

© Kölsch & Altmann GmbH, 2005

Primary & Foreign Keys » CMR

Primary Keys

Primary Keys sind aus der Modellierung relationaler Datenbanken ein Begriff. Worauf wir hier eingehen wollen, ist deren Umsetzung im Zusammenhang mit EJB.

Zunächst hat jede Entity Bean einen Primary Key, entweder vom Container generiert oder vom Entwickler zur Verfügung gestellt. Dadurch unterscheidet sich jede Entity Bean von jeder anderen. Des Weiteren ist jeder Primary Key einer Entity Bean eine Klasse, ob nun eine primitive aus dem JDK (wie z.B. String) oder eine vom Entwickler zur Verfügung gestellte, spezifische Klasse. Eine solche Klasse muss mehrere Eigenschaften besitzen:

- Sie muß serialisierbar sein.
- Sie muß einen – zugänglichen (public) – Default-Konstruktor , sie kann einen weiteren Konstruktor mit den Schlüsselattributen besitzen.
- Ihre – zugänglichen (public) – Attribute repräsentieren die Schlüsselattribute. Diese müssen gleich heißen wie in der Bean.
- Sie muß die Methoden *equals* und *hashCode*, optional die Methode *toString* implementiert haben.

Eine Implementierung einer solchen Schlüsselklasse könnte z.B. folgendermaßen aussehen:

```
import java.io.Serializable;
public class EmployeePK implements java.io.Serializable
{
  public int id;
  public EmployeePK() {
      };
  public EmployeePK(int parId) {
    id = parId;
  }
  public boolean equals (Object obj) {
    if (obj == null) || !(obj instanceof EmployeePK))
      return false;
    else if (((EmployeePK) obj).id == id)
      return true;
    else
      return false;
    }

    public int hashCode() {
      return id;
    }

    public String toString() {
      return String.valueOf(id);
    }

  }
```

© Kölsch & Altmann GmbH, 2005

Eine weitere Besonderheit beim Umgang mit Primary Keys ist der Umstand, dass entsprechende *Primary-Key*-Klassen zum Entwicklungszeitpunkt noch offen gelassen werden können. Statt einer spezifischen Klasse wird zu jenem Zeitpunkt dann einfach die Klasse *Object* angegeben. Beim Deployment muss dann schließlich festgelegt werden, welche Klasse denn nun tatsächlich zum Einsatz kommen soll, sei es nun

- die Klasse *String*
- eine der Wrapperklassen der primitiven Datentypen (z.B. *Integer*, *Float*)
- eine serialisierbare Klasse (wie in obigem Beispiel).

Der Begriff Primary Key ist aus der Welt der *relationalen* Datenbanken entlehnt, während ein Application Server auch mit anderen Datenbanken (v.a. auch sogenannten Legacy-Systemen) und mit ERP-Systemen umgehen können muss. Genau für solche Datenbanken ist das beschriebene Konzept der – zum Entwicklungszeitpunkt – *Unknown Primary Keys* eingeführt worden.

Zusammengefaßt die einzelnen Schritte bei der Verwendung von *Unknown Primary Keys*:

Entwickler	Verwendung der *Primary-Key*-Klasse *Object*
Provider	Rückgabewert von *ejbCreate()* ist *Object* in *ejb-jar.xml* => *Primary Key Typ* = Object
Deployer	*PrimaryKey*-Typ wählen
Deployer	*PrimaryKey*-Klasse zur Verfügung stellen
Deployer	*PrimaryKey*-Klasse in ejb-jar.xml eintragen Belegen im plattformspezifischen Deskriptor (z.B. für BEA Weblogic)

Eine weiterführende Idee dabei ist, dass die Keys auch vom entsprechend angebundenen DBMS generiert werden. Der plattformunabhängige Deskriptor (*ejb-jar.xml*) kann hierbei – auszugsweise – folgendermaßen aussehen:

```
<entity>
  <ejb-name>CourseEJB</ejb-name>

  <local-home>de.kamuc.osv.ejb.course.CourseLocalHome</local-home>
  <local>de.kamuc.osv.ejb.course.CourseLocal</local>
  <ejb-class>de.kamuc.osv.ejb.course.CourseEJB_UPK</ejb-class>

  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <abstract-schema-name>course</abstract-schema-name>

  <cmp-field>
    <field-name>name</field-name>
  </cmp-field>
  <cmp-field>
```

© Kölsch & Altmann GmbH, 2005

```
    <field-name>code</field-name>
  </cmp-field>
      .
      .
      .
  <cmp-field>
    <field-name>coursePK</field-name>
  </cmp-field>

  <primkey-field>coursePK</primkey-field>
      .
      .
      .
</entity>
```

Bei BEA Weblogic sieht der plattformspezifische Deskriptor (*weblogic-cmp-rdbms-jar.xml*) hierfür – auszugsweise – folgendermaßen aus:

```
<weblogic-rdbms-bean>
  <ejb-name>CourseEJB</ejb-name>
  <data-source-name>JNDINameOracleOSV-DB</data-source-name>
  <table-name>course</table-name>
  <field-map>
    <cmp-field>name</cmp-field>
    <dbms-column>name</dbms-column>
  </field-map>
  <field-map>
    <cmp-field>code</cmp-field>
    <dbms-column>code</dbms-column>
  </field-map>
      .
      .
      .
  <field-map>
    <cmp-field>coursePK</cmp-field>
    <dbms-column>coursePK</dbms-column>
  </field-map>
      .
      .
      .
  <delay-database-insert-until>ejbCreate</delay-database-insert-
until>
  <automatic-key-generation>
    <generator-type>ORACLE</generator-type>
    <generator-name>OSV_seq</generator-name>
    <key-cache-size>10</key-cache-size>
  </automatic-key-generation>
</weblogic-rdbms-bean>
```

Das vorige Beispiel dokumentiert auch die Generierung der Primary Keys (*automatic-key-generation*) durch das angebundene DBMS *Oracle*. Umgesetzt wird dies mittels einer Oracle-Sequenz (*osv_seq*) als Generator eindeutiger Nummern.

© Kölsch & Altmann GmbH, 2005

Foreign Keys & CMR

Seit der EJB-Spezifikation, Version 2.0, gibt es auch Container Managed Relationships. Das bedeutet, dass – einfach gesagt – es möglich ist, innerhalb einer Entity Bean auch Attribute anzulegen, die eine Collection von Entity Beans einer anderen – bzw. auch der gleichen – Entity Bean Klasse repräsentieren. Auf DB-Ebene wird hiermit das Konzept der *Foreign Keys* umgesetzt.

Abgebildet wird also die Beziehung von einer zu anderen Entity Beans. Umgesetzt ist dieses Konzept allerdings nur für das Persistenzmodell der *Container Managed Persistence* (CMP). Bei *Bean Managed Persistence* obliegt es nach wie vor dem Entwickler, die entsprechende Navigation zu anderen Beans selbst zu implementieren. Nicht verschweigen wollen wir an dieser Stelle allerdings, dass es aus Performancegründen angebracht sein kann, auf die Möglichkeit der Navigation mittels CMR zu verzichten, weil die Assoziation immer alle Beans einer Entität anspricht. Sollen nur ausgewählte verarbeitet werden, bieten sich Finder- bzw. Select-Methoden der referenzierten Entity Bean selbst an.

Wie von der DB-Modellierung her bekannt, gibt es auch im Zusammenhang mit EJB die üblichen Definitionsmöglichkeiten bzgl. der Kardinalität

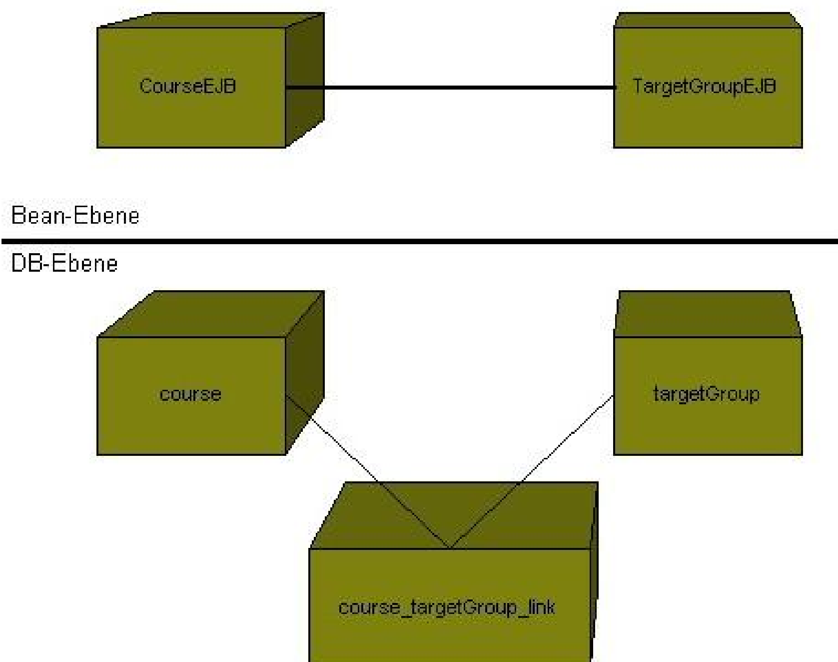
- one-to-one
- one-to-many
- many-to-many

und die Möglichkeit, Einstellungen bzgl. dem *Cascaded Delete* vorzunehmen. Allerdings besteht diese Option lt. Spezifikation nicht bei many-to-many-Beziehungen.

Des Weiteren läßt sich festlegen, ob die Beziehung uni- oder bidirektional angelegt werden soll. Bei einer uni-direktionalen Beziehung läßt sich natürlich nur in eine Richtung navigieren.

Im Folgenden eine Graphik, die bzgl. many-to-many Beziehungen veranschaulicht, dass das EJB-Framework nur die beiden Entity Beans *CourseEJB* und *TargetGroupEJB* kennt, dass dahinter auf der DB-Ebene neben den beiden zugeordneten Tabelle *course* und *targetGroup* auch noch eine weitere Beziehungsentität *course_targerGroup_link* im Spiel ist, mit der die *many-to-many*-Beziehung auf DB-Ebene letztlich erst modelliert werden kann:

© Kölsch & Altmann GmbH, 2005



Der entsprechende plattformunabhängige Deskriptor (*ejb-jar.xml*) hierfür sieht dabei folgendermaßen aus:

```
<ejb-relation>
  <ejb-relation-name>
    CourseLocal-TargetGroupLocal
  </ejb-relation-name>
  <ejb-relationship-role>
    <ejb-relationship-role-name>
      CourseLocal-Has-TargetGroupLocal
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source>
      <ejb-name>CourseEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>targetGroup</cmr-field-name>
      <cmr-field-type>java.util.Collection</cmr-field-type>
    </cmr-field>
  </ejb-relationship-role>
  <ejb-relationship-role>
    <ejb-relationship-role-name>
      TargetGroupLocal-Has-CourseLocal
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source>
      <ejb-name>TargetGroupEJB</ejb-name>
```

```
      </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

Bei BEA Weblogic sieht der plattformspezifische Deskriptor (*weblogic-cmp-rdbms-jar.xml*) hierfür folgendermaßen aus:

```
<weblogic-rdbms-relation>
  <relation-name>CourseLocal-TargetGroupLocal</relation-name>
  <table-name>course_targetGroup_link</table-name>
  <weblogic-relationship-role>
    <relationship-role-name>
      CourseLocal-Has-TargetGroupLocal
    </relationship-role-name>
    <column-map>
      <foreign-key-column>courseFK</foreign-key-column>
      <key-column>coursePK</key-column>
    </column-map>
  </weblogic-relationship-role>
  <weblogic-relationship-role>
    <relationship-role-name>
      TargetGroupLocal-Has-CourseLocal
    </relationship-role-name>
    <column-map>
      <foreign-key-column>targetGroupFK</foreign-key-column>
      <key-column>targetGroupPK</key-column>
    </column-map>
  </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

Abschließend ist beim Einsatz von CMR zu beachten: Durch das Facade-Pattern wird generell schon verhindert, dass *remote* von einer Entity Bean zu einer assoziierten anderen Bean navigiert werden kann. Eine assoziierte Bean *muss* aber auch tatsächlich ein lokales Interface anbieten, d.h. dass Navigation nur innerhalb eines Containers möglich ist.

Bzgl. verschiedener Application Server fällt auf, dass CMRs jeweils auf eigene Art und Weise umgesetzt sind.

- So läßt die **Referenzimplementierung** (Version 1.3.1) hier – entgegen der EJB-Spezifikation – nur unidirektioniale Beziehungen zu, zudem spielt die Reihenfolge der Einträge (*ejb-relationship-role*) im Deployment-Deskriptor eine Rolle (nur in der Vorwärts-Richtung möglich).
- **BEA Weblogic** hat ein ganz eigenes Konzept bzgl. der zu erstellenden Schlüsselklassen. Hier muß die Bean-Klasse abgeleitet werden und in der abgeleiteten Klasse eine *Setter*- und eine *Getter*-Methode für den Primary Key bereitgestellt werden. Sowohl die Klasse also auch die beiden Methoden müssen hierbei *abstract* deklariert werden. Warum diese Klasse nicht auch beim Deployment vom Container selbst mit angelegt wird, ist nicht nachvollziehbar.
- **JBoss** (Version 3.0) gar unterstützt – entgegen der EJB-Spezifikation – in seinen bisherigen Version keine Unknown Primary Keys.

© Kölsch & Altmann GmbH, 2005