Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Untersuchungen von Webanwendungen auf der Basis der J2EE-Umgebung unter z/OS

Diplomarbeit

vorgelegt von Thomas Kumke
Leipzig, im Februar 2005

Für meine Eltern.

Zusammenfassung

Das Institut für Informatik der Universität Leipzig bietet eine Vorlesung mit dem Namen „*z/OS und OS/390* “ an, um den Studenten das Thema der Mainframes näher zu bringen. Des Weiteren werden mehrere z/OS – Rechner unterhalten, um auch praktische Übungen

im z/OS – Umfeld anbieten zu können.
Einer dieser Rechner mit dem Namen YODA verwendet das z/OS – Betriebssystem von IBM in der Version 1.4, in dessen Lieferumfang der WebSphere Application Server 4.0 enthalten ist. Damit wird der professionelle Einsatz von Enterprise Java Beans für die Umsetzung von Webanwendungen ermöglicht.
In dieser Diplomarbeit sollte getestet werden, welche Vorkehrungen getroffen werden müssen, damit eine J2EE – Anwendung auf eben diesem Application Server installiert werden kann. Dazu wurde eine einfache Beispielanwendung (HelloWorld – Anwendung) und eine etwas komplexere Online-bank (WOMBank) implementiert, an deren Beispiel das EJB – Thema erläutert und der Installations-prozess dokumentiert wurde.
Weiterhin sind im Verlauf dieser Arbeit zwei Tutorien entstanden, die in Form von praktischen Übungen den Umgang mit dem WebSphere Application Server erklären und die Einbindung und Anwendung von EJBs in Webapplikationen beschreiben. Die erste Übung geht dabei auf die allgemeinen Grundlagen von Enterprise Java Beans ein und behandelt das Thema der Session Beans. Das zweite Tutorium geht näher auf den Einsatz von Entity Beans ein. Beide bauen auf einer HelloWorld Anwendung auf, die im Laufe der praktischen Übungen weiterentwickelt wird.

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mir mit ihrer Hilfe und Unterstützung die Fertigstellung dieser Diplomarbeit ermöglicht haben.
Mein besonderer Dank gebührt meinem Betreuer Herrn Prof. Spruth, der mir diese Diplomarbeit erst ermöglicht hat und sie stets mit großem Interesse verfolgte und freundlicher Unterstützung begleitete.

Des Weiteren möchte ich mich auch bei Herrn Dr. Herrmann bedanken, der mir bei Fragen und Problemen immer zur Seite stand und mir wegweisende Anregungen für meine Diplomarbeit gab.
Nicht zuletzt danke ich meinem Kommilitonen Georg Müller, der stets dafür gesorgt hat, das der WebSphere Application Server auf dem Yoda – Rechner lief.
Ich möchte mich auch bei meinen Eltern und bei meiner Freundin Franziska Geisler bedanken, die mich die ganze Zeit über moralisch unterstützt haben.

1 Einleitung

1.1 Motivation für den Einsatz von EJB's

Die Entwicklung großer betrieblicher Informationssysteme ist nicht nur gekennzeichnet durch fach-

liche und technische Komplexität, sondern auch durch eine hohe Lebensdauer der Systeme. Deshalb kommen der Wartbarkeit und der Erweiterbarkeit solcher Systeme eine hohe Bedeutung zu. Diesbezüglich werden zur Zeit hohe Erwartungen an die komponentenorientierte Softwareentwicklung geknüpft.
Einen Schritt in diese Richtung unternimmt die Spezifikation der Enterprise Java Beans (EJB) der Firma Sun Microsystems im Rahmen der J2EE – Gesamtspezifikation (Java 2 Enterprise Edition). Durch Komponentenorientierung wird ein hoher Grad an Wiederverwendbarkeit erreicht. Sie ermöglicht es weiterhin, durch den Einsatz von Applikationsservern für alle technischen Aspekte, wie z. B. Datenbankzugriff, Ressourcenverwaltung und Client – Server – Verbindungen, transparente Mechanismen zu verwenden, welche es den Entwicklern ermöglichen, sich voll auf die fachliche Komplexität zu konzentrieren. Die J2EE – Architektur ist zwar noch recht jung (erste Produkte erschienen 1997), aber da die Verbreitung von Java als Programmiersprache und Plattform auch im Serverbereich deutlich zugenommen hat, lohnt es sich für größere Unternehmen J2EE auch in komplexeren Softwareprojekten einzusetzen.

1.2 Ziele

Mit dem Erscheinen des z/OS – Betriebssystems in der Version 1.4 von IBM für Großrechnersysteme, wird der WebSphere Application Server in der Version 4.0 Enterprise Edition angeboten, wobei durch den Zusatz Enterprise Edition schon darauf hingewiesen wird, dass die J2EE – Architektur von SUN komplett unterstützt wird. Ziel dieser Diplomarbeit ist es eine Beispielanwendung zu erstellen, welche auf einer schon vorhandenen Applikation aufbaut, aber den J2EE – Ansatz von SUN verwendet. Die Wahl fiel auf die WOMBank – Applikation, welche eine der DB2 – Beispielanwendungen aus dem Redbook „OS/390 e – Business Infrastructure: IBM WebSphere Application Server 1.2 Customization and Usage“ von Roland Trauner, Denis Gaebler und Bruce Smith ist. Dieses Beispiel wurde schon in der Diplomarbeit „Internet – basierte Anwendungen mit Java und DB2 unter OS/390“ [5] von dem Herrn Ralf Ronneburger in seiner ursprünglichen Form auf dem OS/390 – Rechner der Universität Leipzig mit dem Namen Jedi zum Laufen gebracht. Allerdings ist die Struktur dieser Anwendung sehr einfach. Es wird nur ein Servlet verwendet, um die Anfragen des Users und die dazu nötigen Datenbankzugriffe umzusetzen. In dieser Diplomarbeit wird die WOMBank – Anwendung komplett auf eine J2EE – Architektur umgestellt. Dazu ist es notwendig Entity Beans zu erstellen, welche für die persistente

Datenhaltung verantwortlich sind, Session Beans sollen den Part der Geschäftslogik übernehmen und ein Servlet wird die Eingaben und Anfragen des Nutzers in die nötigen Beanaufrufe umsetzen. Dabei werden die schon vorhandenen HTML – Seiten selbstverständlich weiterverwendet, um das oberflächliche Aussehen sowie die Handhabung der Anwendung nicht zu verändern. Nachdem diese Umstrukturierung der Applikation erfolgt ist und die WOMBank auf dem Entwicklungssystem (Windows XP, IBM WebSphere Application Server 5.1 und IBM DB2 – Datenbank 8.0) lauffähig ist, soll versucht werden, die Anwendung auf dem Einsatzsystem (z/OS – Rechner, WebSphere Application Server 4.0 und IBM DB2 – Datenbank 7.1) zu installieren. Dabei soll eine ausführliche Beschreibung entstehen, die den Installationsprozess nachvollziehbar macht.

2 Grundlagen

2.1 Begriffsklärung

2.1.1 J2EE

Die Java 2 Enterprise Edition (J2EE) von Sun beinhaltet eine Spezifikation für Verteilte Geschäftsapplikationen. Darin ist neben den Komponenten der Geschäftsapplikation, den Enterprise JavaBeans (EJB), auch die Infrastruktur zu deren Ausführung spezifiziert. Die J2EE – Spezifikation zielt auf die Implementierung von mehrschichtigen Client/Server – Anwendungen ab. Mit EJB werden Geschäftsprozesse und Geschäftskomponenten modelliert (z.B. Kunde, Auftrag, Rechnung).

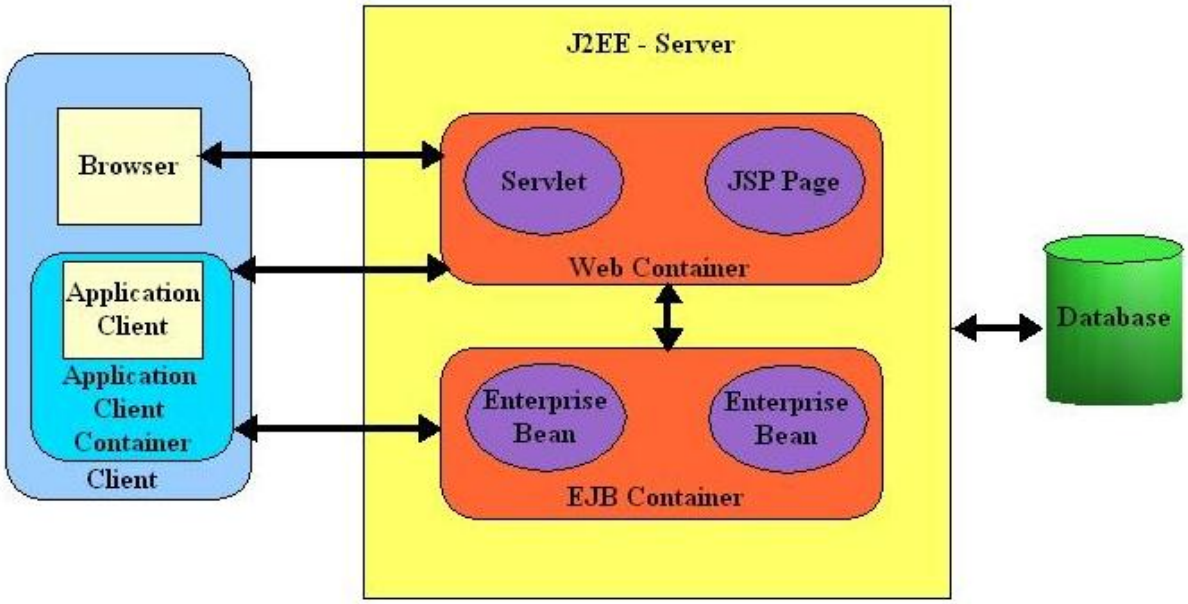


Abbildung 2.1 : Die J2EE – Architektur

Die Infrastruktur beinhaltet Applikationsserver mit sogenannten Containern, in denen die EJB ausgeführt werden. Der Server beziehungsweise der Container interagiert mit den Unternehmenssystemressourcen (z.B. Datenbank) und übernimmt auch die Interaktion mit verteilten Beans in anderen Servern und Maschinen. Weiterhin kontrolliert er die Ausführung von selbst definierten Transaktionen und handhabt Sicherheitseinstellungen. J2EE – Server bieten also typische Middleware – Techniken an, um eine einfache Komponentenprogrammierung zu ermöglichen.
EJB – Programmierer sollen sich größtenteils mit der Lösung der Geschäftsabläufe befassen und wiederverwendbare Komponenten produzieren. Das Idealbild ist, dass diese Komponenten dann in jeglichen nach J2EE spezifizierten Servern laufen sollen. Weiterhin sind als Teil der Infrastruktur auch die Kommunikationsformen der EJB mit den Containern und die Kommunikation zwischen den Containern vorgeschrieben. Ebenso ist die Schnittstelle des Servers zu Clients und Ressourcen weit-

gehend reglementiert.
Weitere Informationen zu dem Thema J2EE finden sich zum Beispiel unter [19] oder [22].

2.1.2 Application Server

Der Begriff „Application Server“ gibt leider kaum Auskunft darüber, was sich hinter dieser Technologie verbirgt. Doch wenn man die verschiedenen, auf dem Markt befindlichen Produkte näher betrachtet, kann man sich mehr unter diesem Schlagwort vorstellen.
Application Server sind Programmpakete, die Techniken wie CORBA, HTTP, Enterprise Java Beans (EJB) und Programmiersprachen wie Java, Perl, C++ unterstützen und mehrere bisher getrennt erhältliche Middleware – Produkte zu einem Produkt zusammenfassen.

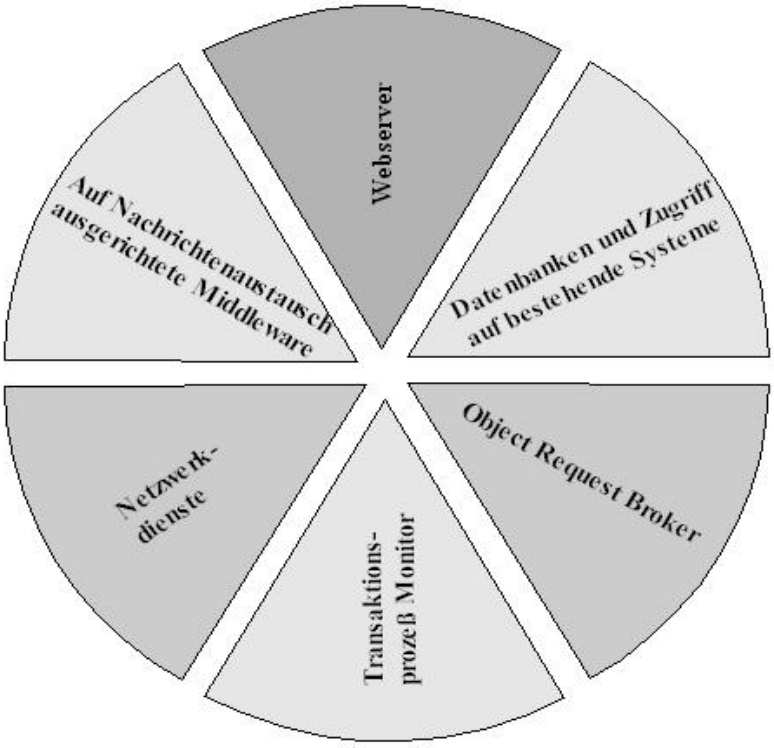


Abbildung 2.2 : Komponenten eines Application Servers

Application Server bieten dem Systemadministrator eine integrierte Bedienoberfläche, mit der die verschiedenen Komponenten und Services installiert und verwaltet werden können. Durch die Zusammenfassung mehrerer bisher getrennter Middleware – Funktionalitäten bilden Application Server das Bindeglied zwischen den Clients auf der einen Seite, die über HTTP und IIOP mit dem Application Server kommunizieren, und der Datenbank auf der anderen Seite, mit der der Application

Server mit Hilfe proprietärer Datenbankschnittstellen oder offener Standards wie ODBC / JDBC kommuniziert. Für weitere Informationen siehe [13] oder [18].

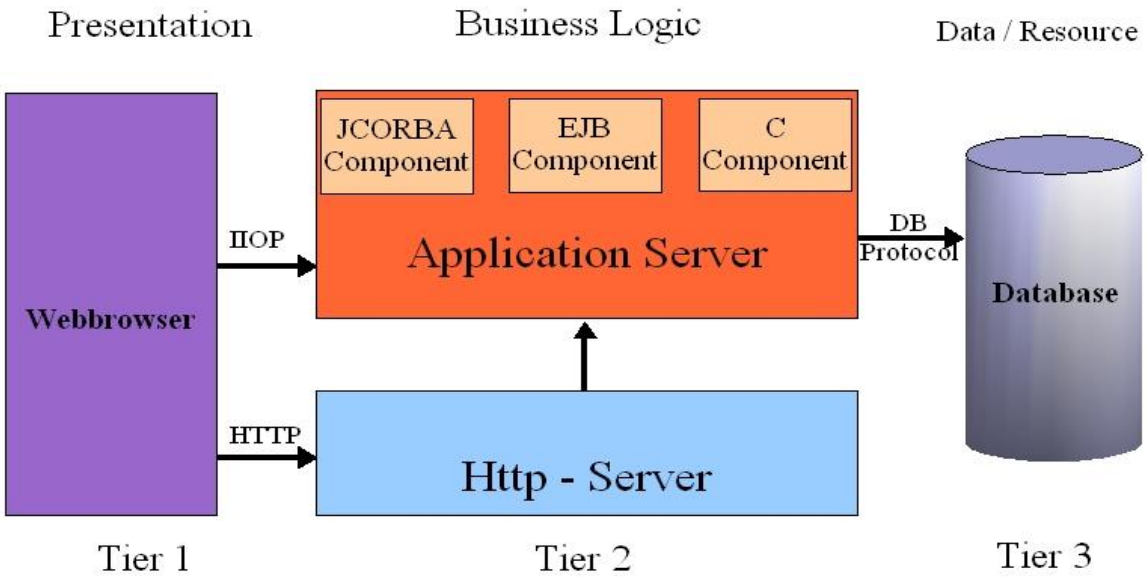


Abbildung 2.3 : Application Server als Bindeglied in einer 3 – Schichten – Architektur

2.2 IBM WebSphere Application Servers

Der IBM WebSphere Application Server hat im Gegensatz zu Konkurrenzprodukten den Vorteil, dass IBM gleichzeitig zu seiner wachsenden Marktstellung im Application Server Bereich auch bei den integrierten Entwicklungsumgebungen für Java mit WebSphere Studio Application Developer eine führende Position eingenommen hat. Damit wird es dem Entwickler ermöglicht, Projekte mit einem modernen und fortschrittlichen Entwicklungswerkzeug zu implementieren und dann diese Applikationen direkt aus dieser Umgebung heraus in ein Test – oder Produktionssystem zu übertragen.

Der WebSphere Application Server ist für fast jedes gängige Betriebssystem erhältlich (neben OS/390, z/OS, Windows NT beziehungsweise XP sowie AIX sind auch Linux und Solaris darunter) Damit wird bei einer Umstellung des Betriebssystems die Portierung der Anwendung erheblich erleichtert oder entfällt ganz. Ein weiterer Vorteil des WebSphere Application Servers ist das Bereitstellen von Schnittstellen zu vorhandenen Software – Plattformen, Businessanwendungen und Middleware wie zum Beispiel SAP, Lotus Domino, CICS und MQSeries. Für diese Diplomarbeit sind vor allem die folgenden Versionen von

Bedeutung.

2.2.1 IBM WebSphere Application Server 1.2

Der WebSphere Application Server in der Version 1.2, welcher auch auf dem Leipziger S/390 – Rechner seinen Dienst verrichtet, ist eine Java Servlet Engine, die als PlugIn für den Domino Go Webserver 5.0 oder für den IBM HTTP Server arbeitet. Die Version 1.2 ersetzt seit OS/390 R5 den IBM WebSphere Application Server 1.1. Der WAS 1.2 unterstützt Java Servlets (API 2.01) und Java Server Pages 0.91 und stellt einige neue Funktionen bereit. So wird zum Beispiel der Connection Manager für JDBC unterstützt, der es ermöglicht, mehrere Datenbankverbindungen für eine Anwendung offen zu halten. Weiterhin können mehrere Instanzen des Application Servers gleichzeitig betrieben werden, wobei jede Instanz eine eigene Konfigurationsdatei besitzt.
Die ursprüngliche Version der WOMBank – Anwendung wurde auf dieser Version des WAS installiert.

2.2.4 IBM WebSphere Application Server 4.0 Enterprise Edition

Der WebSphere Application Server in der Version 4.0 ist nur in der Enterprise Edition erschienen. Er stellt eine J2EE – Laufzeit – Umgebung zur Verfügung und ermöglicht damit den Einsatz von Enterprise Java Beans. Er ist nicht nur für die OS/390 Plattform verfügbar, sondern auch für z/OS. In dieser Version wird das Konzept der Shared Sessions unterstützt. Dabei wird der Sessionzustand einer Webanwendung in einer DB2 – Datenbank zwischengespeichert und kann bei Bedarf wieder ausgelesen werden. Diese Informationen stehen allen anderen Webanwendungen im Sysplex zur Verfügung. Andere Anwendungen im Sysplex können die Sitzungen von Nutzern weiterführen. Die EJB-basierte WOMBank wurde auf dieser Application Server Version installiert.

2.2.5 IBM WebSphere Application Server for z/OS V5

Der WebSphere Application Server für z/OS V5 ist ein Java 2 Enterprise Edition (J2EE) 1.3 – kompatibler Application Server für Webanwendungen, der speziell entworfen wurde, um die einzigartigen Dienste zu nutzen, die von der zSeries Hardware und dem z/OS –

Betriebssystem bereitgestellt werden. Mit dieser Version wird auch die EJB – Spezifikation 2.0 unterstützt, die signifikante Erweiterungen mit sich bringt. Auf diese wird dann in einem späteren Kapitel eingegangen. Ausführlichere Informationen zu den einzelnen Versionen finden sich unter [2],[4],[5],[6] und [23].

2.3 Webanwendungen

2.3.1 Architektur von Webanwendungen

Zur Beschreibung von Client – Server – Architekturen hat sich das Konzept der n – tier – Architektur („tier“ (engl.) = „Schicht“ oder „Etage“) etabliert. Bei einer n – tier – Architektur wird eine Anwendung in mehrere funktionale Bestandteile zerlegt, die dann auf einen Client sowie einen oder mehrere Server verteilt werden. Eine Anwendungsschicht kann dabei nur mit einer anderen Anwendungsschicht kommunizieren, wenn sie unmittelbar an sie angrenzt. Das „n“ bei dem Begriff „n – tier“ legt die Anzahl der Komponenten fest, in die die Anwendung aufgeteilt wird. So ist eine Applikation, die eine 2 – tier – Architektur verwendet, üblicherweise auf zwei Rechnersysteme verteilt, während eine Anwendung mit 3 – tier – Architektur entsprechend drei Rechnersysteme verwendet. Die Zuweisung jeder Komponente zu einem separaten Rechner ist aus verschiedenen Gründen sinnvoll und üblich, nicht aber zwangsläufig notwendig. Auch der Einsatz einer 3 – schichtigen Anwendung auf einem einzigen Rechner ist prinzipiell möglich und wird auch häufig während der Entwicklungsphase einer Applikation betrieben.
Die gewählte Architektur ist entscheidend für die Qualität einer Anwendung und bestimmt maßgeblich dessen Performance, Stabilität, Skalierbarkeit, Flexibilität und Sicherheit. Die üblichen Komponenten von Webapplikationen die bei einer n – tier – Architektur getrennt werden, sind:

- Benutzerschnittstelle
- Geschäfts – bzw. Anwendungslogik
- Datenbasis

Daraus ergeben sich folgende übliche n – tier – Architekturen (siehe dazu [20],[21] und [24]) :

2.3.1.1 2 – tier - Architektur

Die 2 – tier – Architektur repräsentiert die traditionelle Client – Server – Architektur, bei der eine Client-Applikation auf einen entfernten Server zugreift. Die Aufteilung einer Anwendung gemäß einer 2 – tier – Architektur stellt das Minimum für eine Client – Server – Architektur dar. Befinden sich beide Schichten auf dem selben Rechner, wird die Anwendung als eine herkömmliche Desktop – Architektur betrachtet.

Bei einer 2 - tier – Architektur wird die Applikation in eine Datenschicht und eine Präsentationsschicht mit Anwendungslogik aufgeteilt. Der Client implementiert neben der Benutzerschnittstelle auch die gesamte Fachlogik, sowie die zum Zugriff auf den Datenbankserver notwendige API (man spricht in diesem Zusammenhang deshalb auch von „Fat Client"). Bis auf den Datenbestand existiert die Applikation damit vollständig auf der Client – Maschine. Der Server dient lediglich zur Bereitstellung der Daten.

Ein Beispiel für eine auf 2 – tier – Architektur basierende Anwendung ist ein Java – Applet mit JDBC – Treibern auf Client – Seite und einem Datenbank – Management – System (DBMS) auf Serverseite. Aber auch eine Webapplikation mit rein statischen Inhalten, bestehend aus Webbrowser und Webserver, stellt eine 2 – tier – Architektur dar.

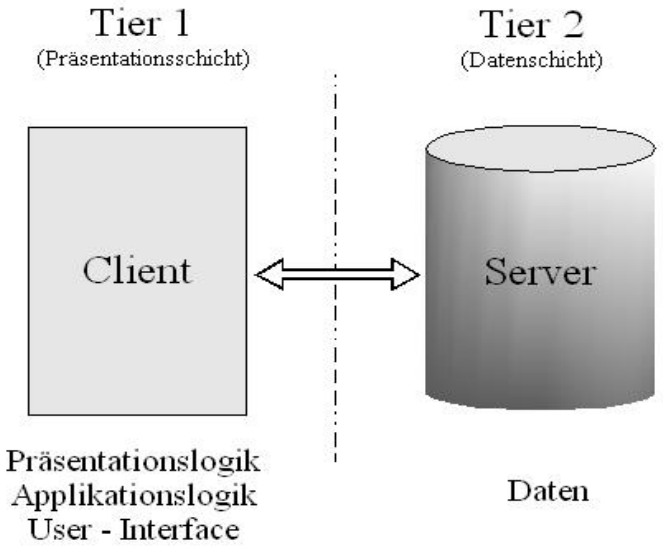


Abbildung 2.4 : 2 – tier – Architektur

Die Vorteile einer solchen Architektur sind die einfache Realisierung, sowie ihre weite Verbreitung und Unterstützung. Dennoch ist die Verwendung der 2 – Schichten – Architektur für Webapplika-

tionen nicht ratsam, da die nicht vorhandene Trennung zwischen Benutzerschnittstelle und Anwendungslogik zu einer Reihe von Nachteilen führt. Diese sind im Einzelnen :

- Monolithische, undurchsichtige Struktur der Client-Applikation
- Schlechte Skalierbarkeit und Wiederverwendbarkeit
- Schlechte Performance (diese sinkt deutlich bei zunehmender Benutzerzahl)
- Schlechte Erweiterbarkeit der Logik
- Änderungen der Fachlogik oder des Datenmodells erfordern Änderungen bei jedem einzelnen Client
- Die nicht vorhandene Möglichkeit der Trennung der Anwendungskomponenten vom Client stellen ein erhöhtes Sicherheitsrisiko dar

2.3.1.2 3 – tier – Architektur

Bei einer 3 – tier – Architektur wird die Anwendung im Vergleich zu einer zweischichtigen Architektur um eine weitere Ebene ergänzt und damit in insgesamt drei logische Schichten aufgeteilt. Diese sind die Präsentationsschicht, die Applikationsschicht und die Datenschicht.

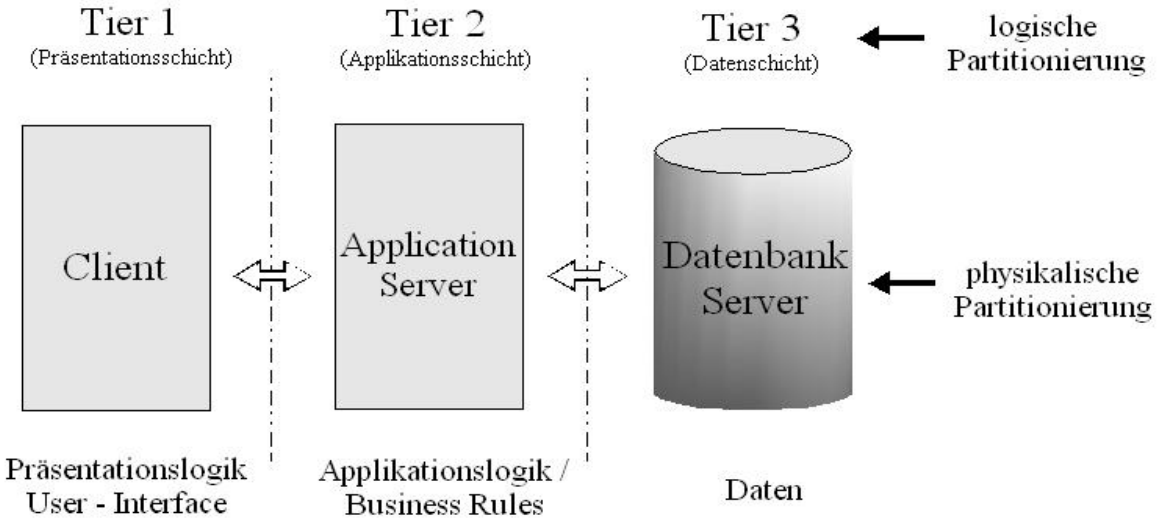


Abbildung 2.5 : 3 – tier – Architektur

Der Client, der nun nur noch die Präsentationsschicht realisiert, greift dadurch nicht mehr selbst auf die Datenbasis zu, sondern auf einen zur Umsetzung der Applikationsschicht zuständigen Applikationsserver, der wiederum mit dem für die Datenschicht zuständigen Datenhaltungssystem

kommuniziert. Der Client wird damit um die gesamte Anwendungslogik entlastet und realisiert nur noch die Benutzerschnittstelle. Er ist dadurch klein und wird deshalb als „Thin Client“ bezeichnet.

Der neu hinzugekommene Applikationsserver kapselt die gesamte komplexe Fachlogik und realisiert eine API zum Zugriff auf die Daten des Datenhaltungssystems. Der Einsatz mehrerer Applikations - server ermöglicht durch Verwendung von Load – Balancer (wie z.B. Transaction Processing (TP) - Monitore) die Verteilung der Client – Zugriffe auf verschiedene parallel geschaltete Server. Bei Bedarf können damit weitere Applikationsserver hinzugefügt werden, ohne dass eine Änderung der Architektur notwendig ist. Oft wird die Applikationsschicht auch durch einen reinen, nur für die Entgegennahme und Beantwortung der Clientanfragen zuständigen Webserver und einen (oder mehre- ren) Applikationsserver(n) realisiert, die untereinander kommunizieren. Aufgrund der physikalischen Aufteilung auf vier Rechner wird bei solch einer Realisierung teilweise auch von einer 4 – tier – Architektur gesprochen.

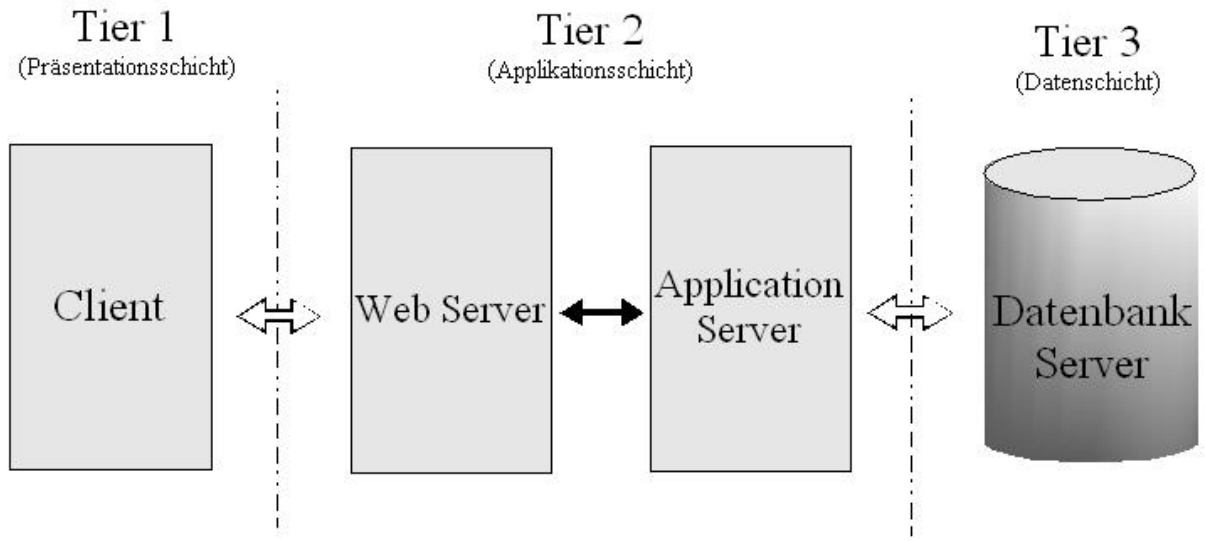


Abbildung 2.6 : 3 – tier – Architektur mit zusätzlichem Webserver

Der Datenbank – Server schließlich dient nur der Datenhaltung. Er kapselt die von der Anwendung benötigten Daten und stellt diese dem Applikationsserver bereit. Eine Replikation des Datenbank – Servers zur Lastverteilung ist möglich. Die Vorteile einer 3 – tier – Architektur liegen damit auf der Hand :

- Beliebig hohe Skalierbarkeit durch Einsatz mehrerer Applikationsserver

- Hohe Performance durch Verteilung der Aufgaben auf verschiedene Server
- Hohe Wiederverwendbarkeit der Anwendungslogik
- Höhere Sicherheit, da keine Umgehung der Fachlogik auf Client-Seite möglich ist und der Client nicht mehr direkten Zugriff auf die Datenbank besitzt
- Änderungen der Fachlogik erfordern nur noch Änderungen am Applikationsserver

Demgegenüber stehen allerdings auch eine deutliche höhere Komplexität der Applikation und damit ein hoher Aufwand beim Aufbau einer 3 – Schichten – Architektur. Wie vormals schon erwähnt, muss die logische Partitionierung einer Anwendung nicht zwangsläufig mit ihrer physikalischen Partitionierung übereinstimmen. Aufgrund der einfachen Skalierbarkeit von 3 – tier – Architekturen können einzelne Schichten zusammen auf einem Rechner betrieben werden, solange die Benutzerzahl gering ist und erst bei höherem Leistungsbedarf getrennt werden.

2.3.1.3 n – tier – Architektur

Eine n – tier – Architektur ist nichts anderes als eine 3 – tier – Architektur, bei der die Applikationsschicht in eine unbeschränkte Anzahl weiterer funktionaler (nicht unbedingt physisch getrennten) Schichten unterteilt wird. Sie könnte beispielsweise aus folgenden Schichten bestehen:

- eine Benutzerschnittstelle zur Interaktion mit dem Benutzer,
- eine Präsentationslogik die bestimmt was die Benutzerschnittstelle darstellt und wie auf Anforderungen vom Benutzer reagiert wird,
- eine Applikationslogik welche die Geschäfts – bzw. Anwendungsregeln abbildet,
- Infrastruktur Services welche zusätzliche Funktionalitäten anbieten, die benötigt werden (Nachrichtenverkehr, Transaktionssicherheit und so weiter)
- eine Datenschicht, welche die Daten verwaltet.

2.3.2 Clientseitig und Serverseitig

Das World Wide Web wurde Anfang der 90er Jahre als Informationssystem für statische Dokumente entwickelt. Die Funktionsweise ist sehr einfach, wie das folgende Beispiel zeigt. Ein Benutzer fordert ein Dokument an, der entsprechende Webserver liefert den Inhalt der zugehörigen Datei an den

Webbrowser und dieser stellt das Dokument visuell dar. Das Potenzial des WWW ist damit bei weitem noch nicht ausgeschöpft. Die beiden Endpunkte der Kommunikation, Webbrowser und Webserver können viel mehr leisten, als in dem gerade dargestellten Szenario erkennbar ist. Die Realisierung komplexer Anwendungen wie Internethandel oder Zugang und Pflege von großen Produktkatalogen über das WWW basiert unter anderem darauf, dass

- Webbrowser Benutzerdaten aufnehmen und analysieren können und
- Webserver Dokumente dynamisch erzeugen und mit anderen Anwendungen kommunizieren können.

Techniken zur Realisierung webbasierter Anwendungen werden in Abhängigkeit ihres Ausführungsortes in zwei Gruppen aufgeteilt: Serverseitige und clientseitige Techniken. Serverseitige Anwendungen sind für die Erzeugung bzw. Bereitstellung der Daten verantwortlich. Clientseitige Anwendungen sind für die Darstellung der Daten und die Benutzerinteraktionen zuständig. CGI – Programme sind Beispiele für die erste Gruppe und Java Applets für die zweite. Die Entwicklung der einzelnen Techniken erfolgte schrittweise, treibende Kraft war der enorme Erfolg des WWW im kommerziellen Bereich. Siehe dazu auch [3].

2.3.2.1 Clientseitige Lösungen für Webanwendungen

Clientseitige Lösungen sind vor allem Java – Applets, JavaScript und vor allem Anwendungen im firmeninternen Gebrauch. JavaScript stellt dabei aber ein Sonderfall dar, da es dynamische Effekte bei der Verarbeitung von Daten erzeugen kann, aber der Datenaustausch mit dem Server wird nicht ermöglicht. Weiterhin kann der Einsatz von JavaScript durch die Browsereinstellungen unterbunden werden. Dies geschieht oft durch den Systemadministrator, damit dieser eine bessere Kontrolle über den Nutzer erhält.
Anwendungen, die speziell für die Benutzung im firmeninternen Umfeld vorgesehen sind, müssen auf jedem Client installiert und gewartet werden. Sie sind somit nicht für eine große Benutzerzahl ausgelegt. Des Weiteren benötigen diese Spezialanwendungen Speicherplatz und Rechenleistung auf der Clientseite, da die Informationen aus der Datenbank erst hier aufbereitet werden. Eine Bereitstellung für eine große Anzahl von Plattformen und Nutzern ist vom Kosten – und Entwicklungsstand nicht vertretbar, da der Aufwand für die Erstellung einer solchen Anwendung für verschiedene Platt-formen den Nutzen übersteigen würde.

Genau wie die Spezialanwendungen benötigen Java – Applets auf der Clientseite Speicherplatz und Rechenleistung und sind auch nicht auf allen Betriebssystemen lauffähig. Voraussetzung für den Betrieb eines Applets ist ein auf dem Client installierter Browser, der die verwendete Java – Version unterstützt.
Eine wesentliche Entlastung des Servers wird aber durch diese clientseitigen Lösungen nicht begünstigt, da jeder Client seine eigenen Daten anfordert und selbst bei Änderungsoperationen mit dem Server kommuniziert. Weiterhin wird sehr viel Zeit vom Client darauf verwendet, die Datenbankverbindungen zu öffnen und zu schließen, damit ist ein Pooling der Verbindungen nicht möglich. Ein weiteres Problem stellt die Sicherheit dar. Es besteht die Gefahr, dass die Verbindungen zwischen Client und Server von einem Dritten abgehört werden und eine Verschlüsselung auf Client – beziehungsweise Serverseite wäre wiederum mit mehr Aufwand verbunden.

2.3.2.2 Serverseitige Lösungen für Webanwendungen

Wie der Name schon sagt, werden die Informationen bei einer serverseitigen Lösung komplett auf dem Server aufbereitet und danach an den Client geschickt, der die Anfrage gestellt hat. Dabei kann die Art des Dokuments dynamisch an die Fähigkeiten des Clients angepasst werden (HTML – Seiten, PDF – Dokumente).
Im Normalfall stellen viele Clients gleichzeitig Anfragen an den Server und die Wartezeit für einen Aufruf möglichst gering zu halten, muss der Server über die entsprechenden Ressourcen verfügen. Dies könnte man als Nachteil werten, allerdings muss bei Performanceproblemen nur der Server aufgerüstet werden.
Gängige serverseitige Lösungen um den Inhalt von Internetseiten dynamisch zu präsentieren sind unter anderem Active Server Pages (ASP), wichtigster Wettbewerber für die JSP – Technologie von Microsoft, Common Gateway Interface (CGI), Java Server Pages (JSP) und Java – Servlets. Auf diese Lösungen wird im nächsten Kapitel detailiert eingegangen, wobei besonderen Wert auf die hier nicht genannten Enterprise Java Beans gelegt wird, da diese nicht unbedingt die dynamische Präsentation erleichtern wohl aber die Datenbeschaffung. Um diese Informationen dann wieder an den Client angepasst darzustellen, können Java – Servlets und Java Server Pages zum Einsatz kommen, auf die im folgenden Kapitel auch kurz eingegangen wird.

3 Serverseitige Technologien für Webanwendungen

3.1 Common Gateway Interface (CGI)

Der erste Schritt in Richtung dynamisch erzeugter Dokumente bildete das Common Gateway Interface (CGI). Ein CGI – Programm realisiert den interaktiven Zugriff auf eine Informationsquelle, so dass die Information für einen Web – Client wie eine Datei auf einem Webserver erscheint. Auf diese Art können leicht komplexe Anwendungen erstellt werden wie Zugriff auf Datenbanken, Datenvisualisierung, Datenerfassung und so weiter. Ein CGI – Programm ist ein ausführbares Programm. Bei jeder Anfrage startet der Webserver dieses in einem separaten Prozess. Zur Übergabe von Daten an das Programm belegt der Server Umgebungsvariablen mit festgelegten Namen. Diese werden dann von dem Programm gelesen. Das Programm schreibt das erzeugte HTML – Dokument einfach auf die Standardausgabe, der Webserver liest die Daten von dort ein und schickt sie unverändert an den anfragenden Client. Die Popularität von CGI – Anwendungen rührt unter anderem daher, dass sie fast mit jeder Programmiersprache erstellt werden können. Die Sprache muss nur den Zugriff auf Umgebungsvariablen und die Standardeingabe bzw. – ausgabe unterstützen. Existierende Anwendungen können dadurch auch leicht auf verschiedene Webserver portiert werden. CGI – Programme sind hauptsächlich für einfache Anwendungen geeignet.
Werden die Anwendungen umfangreicher und komplexer, so stößt man bald an die Grenzen von CGI. Viele Vorgänge laufen bei einer Web – Anfrage innerhalb des Webservers ab, das Konzept des CGI ermöglicht aber nicht den Zugang zu diesen internen Prozessen. Beispiel hierfür sind Zugangskontrolle, Authentifizierung, die Zuordnung von URLs zu Pfadnamen und so weiter. Webserver erzeugen für jede CGI – Anfrage einen eigenen Prozess. Darin liegt der wohl größte Nachteil dieser Technik. Der mit jeder Anfrage einhergehende Verbrauch von Betriebssystemressourcen kann sich negativ auf die Verfügbarkeit des Servers auswirken. Des Weiteren wird dadurch die Realisierung von sitzungsorientierten Anwendungen, also die Umsetzung eines Transaktionskonzeptes, sehr erschwert. Weitergehende Informationen zu dem Thema CGI finden sich unter [3],

3.2 Java Servlets und Java Server Pages

3.2.1 Java Servlets

Mit der Servlet API bietet Sun eine Anwendungsprogrammierschnittstelle für serverseitige Java – Programme. Die Servlet API ist eine Standard Java-Erweiterung und sie stellt ähnliche Zugriffsmöglichkeiten wie bei CGI zur Verfügung. So ist der Zugriff auf Umgebungsvariablen möglich und es existieren Datenströme für die Anfrage und die Antwort. Zusätzlich bietet wird eine Unterstützung für Cookies und ein Session – Management.
Ein Servlet ist eine, auf Java basierende, Webkomponente. Durch die Verwendung der Java – Technologie ist die volle Plattformunabhängigkeit gegeben. Dies ermöglicht daher einen flexiblen Einsatz auf allen gängigen Plattformen.
Servlets sind dazu gedacht, den Inhalt einer Website, dynamisch zu generieren. Sie sind rein serverseitige Komponenten, und bilden daher eine mächtige Erweiterung der Serverfunktionalität. Im Gegensatz zu CGI wird beim Servlet – Ansatz bei einer Anfrage für ein Servlet nicht jedes Mal ein eigener Prozess erzeugt, sondern nur ein Thread. Daher sind bei konkurrierenden Anfragen keine Prozesswechsel des Betriebssystems notwendig und es tritt dadurch eine deutliche Geschwindigkeitsverbesserung ein.

Der Servletcontainer ist ein Teil des Web – oder Applikationsservers. Dieser stellt die Netzwerkdienste zur Verfügung, welche benötigt werden, um Client – Anfragen (request) und die von den Servlets erzeugten Antworten (response) zu verarbeiten. Der Container ist auch für die Verwaltung der Servlets verantwortlich, und führt diese durch ihren Lebenszyklus.
Der Servlet Container kann entweder direkt in den Webserver integriert werden, oder als Erweiterung in der Server eingebettet werden. Der Servlet Container muss das HTTP – Protokoll unterstützen. HTTPS (HTTP über SSL) sollte unterstützt werden, ist aber laut Spezifikation keine Voraussetzung. Der Servlet – Container sollte mindestens die Version 1.0.3 des HTTP – Protokolls implementieren. Es wird aber mit Nachdruck empfohlen HTTP in der Version 1.1.4 ebenfalls zu unterstützen.

An einem einfachen Beispiel soll nun gezeigt werden, wie eine Anfrage an den Server durchgeführt wird. Der Client baut mit einem Browser die Verbindung zum Webserver auf, und schickt dabei einen HTTP – Anfrage. Dieser Request wird vom Webserver empfangen und an den Servlet Container

weitergereicht. Dieser entscheidet nun auf Grund der Konfiguration seiner Servlets, an welches Servlet der Request weitergeleitet wird. Dabei wird jeweils ein Objekt für den Request und den Response, welcher später als Antwort gesendet werden soll, erzeugt. Diese beiden Objekte werden nun dem Servlet übergeben. Das Servlet wertet nun die Daten aus dem Request aus. Dabei kann es feststellen, von wem die Anfrage kommt, wie eventuell HTTP POST Parameter gesetzt sind, oder andere relevante Daten. Aus den gewonnenen Daten modifiziert das Servlet nun das Response-Objekt. Nach der vollständigen Erzeugung des Response – Objekts sorgt der Servlet Container dafür, dass der Response an den Client zurückgesendet wird. Anschließend wird die Kontrolle wieder dem Webserver übergeben.

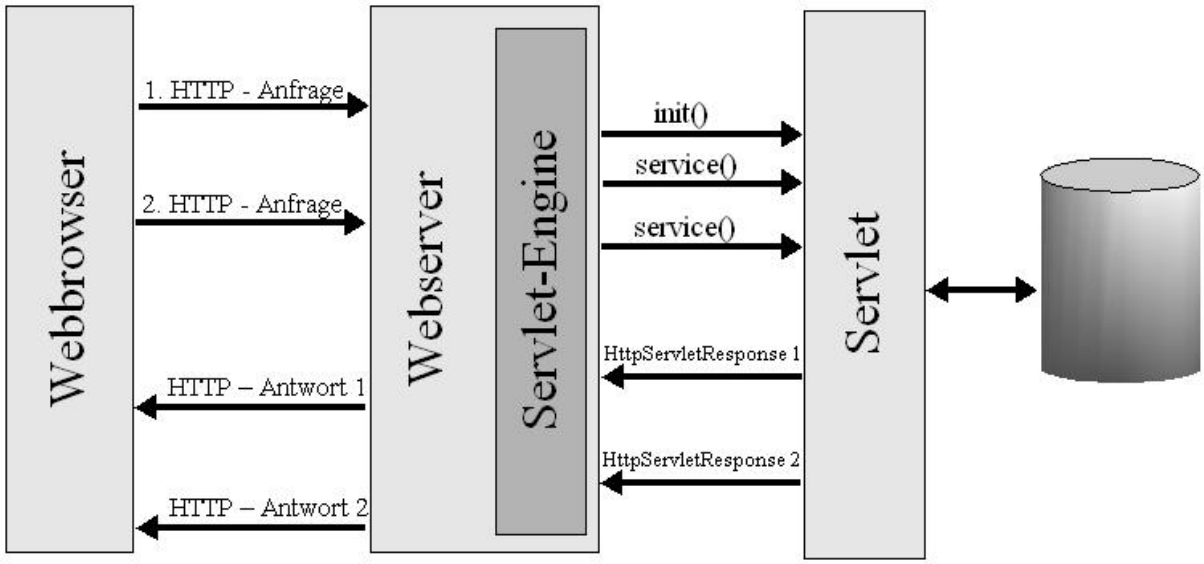


Abbildung 3.1 : Servlet – Kommunikation

Ein Servlet besitzt einen genau definierten Lebenszyklus, dessen Ablauf vom Servlet Container gesteuert und überwacht wird. Im wesentlichen wird dieser Zyklus durch die Methoden des API ausgedrückt. Dabei muss jedes Servlet das javax.servlet.Servlet Interface direkt, oder indirekt über die abstrakten Klassen GenericServlet oder HttpServlet, implementiert haben. Die 3 wichtigsten Methoden dieser Interfaces sind die Methoden:

- init()
- service()
- destroy()

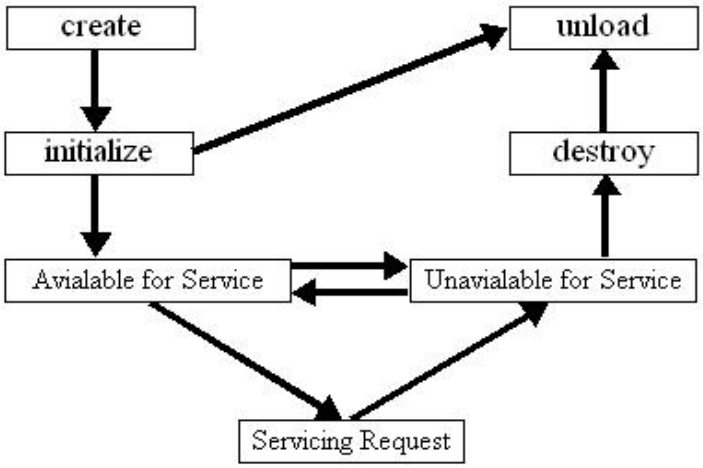


Abbildung 3.2 : Der Lebenszyklus eines Servlets

Wird ein Servlet benötigt, welches noch nie verwendet wurde, so muss dieses vom Servletcontainer zuerst referenziert werden. Das heißt, der Container lädt das Servlet dynamisch in den Speicher.
Dabei gibt es zwei Varianten, wann ein Servlet geladen wird.
1. Das Servlet wird beim Starten des Servers schon geladen und steht von Beginn an im Speicher zur Verfügung.
2. Der Servlet – Container lädt das Servlet in den Speicher wenn eine HTTP – Anfrage an eine URL gestellt wird. Dieser weiß, welches Servlet an diese URL gebunden ist und lädt das entsprechende Servlet. Dieser Vorgang wird jedoch nur bei der ersten Anfrage durchgeführt.
Bevor ein Servlet für die Verwendung zur Verfügung steht, muss es initialisiert werden. Dies geschieht mit der Methode *init()*. Dabei wird dem Servlet ein Objekt von Typ *ServletConfig* übergeben. Mit diesem Objekt kann das Servlet per Name auf Initialisierungsdaten der Webapplikation zugreifen.
Konnte das Servlet ordnungsgemäß initialisiert werden, steht es nun bereit, einen HTTP – Request entgegenzunehmen. Damit dies geschehen kann, wird vom Servlet Container die Methode *service()* aufgerufen. Dieser Methode wird das HTTP Request – Objekt und das HTTP Response – Objekt als Parameter übergeben. Ein Servlet kann während der Bearbeitung einer Anfrage entweder eine ServletException oder eine UnavailableException verursachen.
Eine ServletException tritt dann auf, wenn ein Fehler bei der Bearbeitung der HTTP – Anfrage aufgetreten ist. Der Servlet Container hat nun zu entscheiden, ob die Verarbeitung eines Request abgebrochen oder erneut gestartet wird.
Eine UnavailableException signalisiert, dass das Servlet im Moment nicht in der Lage ist einen

HTTP-Request entgegen zu nehmen. Dabei unterscheidet man zwischen 2 Varianten, permanent und vorübergehend. Bei der ersten Variante ist ein Fehler aufgetreten, der es für das Servlet unmöglich macht jemals wieder einen HTTP – Request zu bearbeiten. Der Servlet Container hat nun die Aufgabe die Methode service() zu beenden und das Servlet aus dem Adreßraum zu entfernen.
Im zweiten Fall wird dem Servlet Container gezeigt, dass er für die Zeit dieses Zustandes dem Servlet keine Anfrage mehr zuordnen soll. Jeder Request der in dieser Zeit abgelehnt wird, muss mit einem *SERVICE UNAVAILABLE (503)* Response beantwortet werden.
Bevor ein Servlet aus dem Adressraum entfernt werden soll, muss vorher die Methode *destroy()* aufgerufen werden. Diese Methode gibt dem Servlet die Chance alle Ressourcen, die vom Servlet genutzt wurden, frei zu gehen. Dabei kann es sich zum Beispiel um Datenbankverbindungen oder Hintergrund Threads handeln. Nachdem alle Aufräumarbeiten beendet worden sind, kann der Speicherbereich freigegeben werden und die Referenz auf das Servlet fallen gelassen werden.

Trotz der ausgesprochen guten Fähigkeiten der Servlets, gibt es dennoch ein herausragendes Defizit. Servlets genügen leider nicht der Anforderung Information und Darstellung ordentlich zu trennen. Darum war man bestrebt, eine Technologie, aufbauend auf Servlets, zu entwickeln, die dieser Vorgabe gerecht wird. Aus diesen Überlegungen heraus entstanden JSPs mit ihren Tag – Libraries.
Weiterführende Informationen zu Java Servlets finden sich unter anderem in [3],[11],[16],[17] und [22].

3.2.1 Java Server Pages

Java Server Pages sind im Aufbau vergleichbar mit HTML-Dokumenten. Sie bestehen jedoch aus zwei Teilen, einem HTML – Text sowie beliebig vielen, darin eingebetteten Anweisungen an den JSP – Server . Zum Einfügen von JSP – Anweisungen dienen spezielle Tags, die den aus HTML bekannten Tags größtenteils gleichen. HTML – Tags behalten ihre normale Funktion und dienen auch hier dem Formatieren des anzuzeigenden Dokuments. Im Gegensatz zu HTML – Tags haben JSP – Tags nicht immer direkten Einfluss auf die Ausgabe. Sie werden nicht zum Client gesandt und machen sich nur durch Seiteneffekte serverseitig ausgeführter Aktionen bemerkbar. Der Hauptzweck dieser Aktionen besteht im Allgemeinen in der Umsetzung einer Anwendungslogik. Zu den am häufigsten benötigten Aktionen gehören Datenbankoperationen, wie das Auslesen oder Einfügen von Datensätzen oder die Prüfung von Zugangsberechtigungen.

Um Wünsche des Benutzers berücksichtigen zu können, ist es für ein serverseitig ausgeführtes Programm notwendig, Daten vom Client zu erhalten. Die zur Datenaufnahme notwendige Funktionalität wird bereits durch die Seitenbeschreibungssprache HTML zur Verfügung gestellt. Anfragen aus HTML - Dokumenten heraus können mit Hilfe von Formularen generiert werden. Das dazu verwendete <form> – Tag erhält als Zielanwendung die URL einer JSP – Anwendung. Diese kann wiederum alle im Formular enthaltenen Daten direkt übernehmen, verarbeiten und daraus eine passende Antwort generieren.

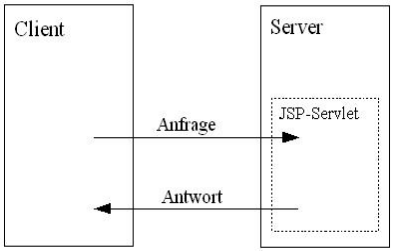
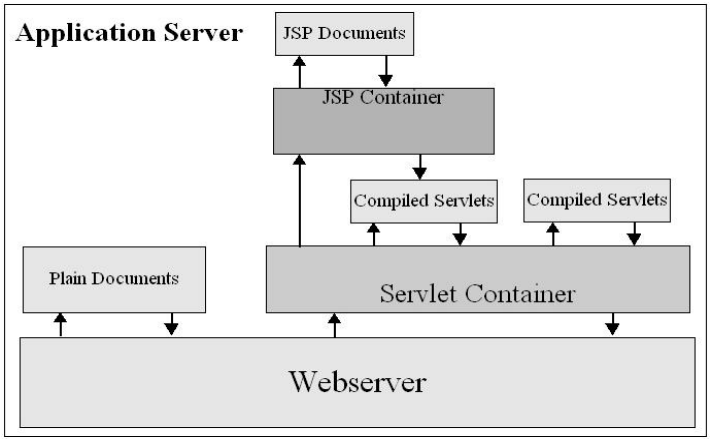


Abbildung 3.3 : Interaktion zwischen Client und Server

Der JSP – Container baut in einem Application Server auf dem Servlet – Container auf, was durchaus Vorteile mit sich bringt. Die Entwickler von Webapplikationen müssen nur eine einzige Haupt – API kennen, nämlich die Servlet – API. Die JSP – API ist ein relativ kleine Erweiterung, die schnell erlernt werden kann und einem Servlet – Entwickler ermöglicht, in kurzer Zeit mit JSP vertraut zu werden. Durch die Trennung der Schichten ergeben sich weitere Vorteile. Ein JSP – Container baut vollständig auf der Servlet – API auf. Sie muss keinen Code von dem Servlet - Container reproduzieren können. Dies verringert die Komplexität bei der Implementierung einer JSP – Anwendung. Außerdem können die JSP – und die Servletschicht einer Webapplikation unabhängig voneinander getestet werden. Durch die Ähnlichkeit beider Container können Entwickler von Webapplikationen Code aus Servlets und anderen Klassen in JSPs und umgekehrt ohne große Mühe wieder verwenden.

Abbildung 3.4 : Zusammenhang zwischen JSP – und Servletcontainer in einem Application Server



Der Webserver erkennt die Dateiendung .jsp innerhalb der URL. Daher ist klar, dass die angeforderte Ressource eine Java Server Page ist und somit von der JSP – Maschine behandelt werden muss. Die JSP-Seite wird dann in eine Java – Klasse transformiert, aus der dann ein Servlet kompiliert wird. Diese Phase der Transformation und Kompilation fällt nur an, wenn die JSP-Seite das erste Mal aufgerufen wird oder sich der Inhalt (Code) ändert. Beim ersten Aufruf einer Java Server Page stellt man deshalb eine leichte Verzögerung fest, welche auf das Kompilieren zurück-zuführen ist. Bei jedem weiteren Aufruf tritt diese Verzögerung nicht mehr auf, weil jede Anfrage einer umgehend beantwortet wird, weil die Anfrage direkt an das fertige Servlet im Speicher des Servers weitergeleitet wird. Für weitere JSP – Informationen siehe auch [3],[11] und [17].

3.3 Enterprise Java Beans (EJB)

Mit der Einführung von RMI und JavaBeans als Teil der Core – APIs wurde Java um ein standardisiertes Framework für verteilte Objekte und um ein Komponentenmodell erweitert. Die Enterprise JavaBeans – Architektur (EJB) stellt auf dieser Grundlage ein standardisiertes Modell für verteilte Komponenten zur Verfügung.

Eine EJB – Komponente verfügt über die Möglichkeit entfernter Zugriffe wie ein RMI oder CORBA Objekt in dem Sinne, dass sie als Remote – Objekt exportiert werden kann, wozu RMI/IIOP verwendet wird. Und eine EJB – Komponente ist immer auch eine JavaBeans – Komponente, denn sie hat Eigenschaften, die durch Introspektion untersucht werden können, und sie verwendet die JavaBeans – Konventionen bei der Definition von Zugriffsmethoden für ihre Eigenschaften. Allerdings verbirgt sich hinter EJBs viel mehr als die Summe dieser Teile. Die EJB – Architektur bietet ein Framework, mit dem Entwickler von Enterprise Beans leicht die Fähigkeiten zur

Transaktionsverarbeitung, Sicherheit, Persistenz und zum Ressourcen-Pooling nutzen können, die durch eine EJB-Umgebung bereitgestellt werden.

Enterprise JavaBeans können sinnvollerweise überall dort eingesetzt werden, wo verteilte Objekte benötigt werden. Von überragendem Nutzen sind sie aber dann, wenn Sie die Natur der EJB – Objekte als Komponenten und auch die anderen Dienste nutzen können, die EJB-Objekte relativ einfach zur Verfügung stellen können, zum Beispiel die Transaktionsverarbeitung und die Persistenz. Ein gutes Beispiel dafür ist eine Online – Banking – Anwendung: Ein Nutzer sitzt zu Hause und möchte sich mit allen ihren Bankkonten verbinden, gleichgültig wo und bei wem sich diese befinden, und sie alle im Zusammenhang und in einer angenehmen Benutzeroberfläche vorfinden. Die EJB – Komponentenarchitektur ermöglicht es den verschiedenen Finanzinstituten, Benutzerkonten als unterschiedliche Implementierungen eines gemeinsamen Interfaces *Account* genau so zu exportieren, wie Sie dies mit anderen APIs für verteilte Objekte täten. Aber da die Account - Objekte zugleich EJBs sind, kann man die Transaktionsfähigkeiten des EJB – Servers dazu nutzen, die Account – Objekte als transaktionale Komponenten zu implementieren, so dass der Client mehrere Kontenoperationen innerhalb einer einzigen Transaktion durchführen und sie dann alle entweder mit einem Commit oder einem Rollback abschließen kann. Bei Finanzanwendungen kann dies ein entscheidender Vorteil sein, insbesondere wenn Sie etwa sicherstellen müssen, dass eine Überweisung stattgefunden hat, bevor eine Abhebung in Auftrag gegeben wird. Die Transaktionsunterstützung von EJB stellt sicher, dass, wenn während des Transfers ein Fehler auftritt und eine Exception ausgelöst wird, für die gesamte Transaktion ein Rollback durchgeführt werden kann und die clientseitige Anwendung Ihnen den Grund dafür mitteilen kann. Und natürlich kann eine Anwendung wie diese auch die Vorteile aller anderen Komponentendienste nutzen, die in Zusammenhang mit EJB verfügbar sind.

Das EJB – Komponentenmodell isoliert Anwendungen und größtenteils Beans von den Einzelheiten der in der Spezifikation enthaltenen Komponentendienste. Ein Vorteil dieser Trennung ist die Möglichkeit, dieselbe Enterprise - Bean unter verschiedenen Bedingungen zu betreiben, die von der jeweiligen Anwendung vorgegeben werden. Die Parameter, die zur Kontrolle der transaktionalen Natur, ihrer Persistenz, des Ressourcen-Pooling und des Security – Managements verwendet werden, sind in gesonderten *Deployment Deskriptoren* beschrieben und befinden sich nicht in der Bean – Implementierung oder im Code der Client-Anwendung. Wenn also eine Bean innerhalb einer verteilten Anwendung in Betrieb genommen wird, können die Eigenschaften der Laufzeitumgebung

(Client-Belastungsgrenzen, Datenbank – Konfiguration und so weiter) in den Deployment – Optionen der Bean berücksichtigt und entsprechend eingestellt werden.

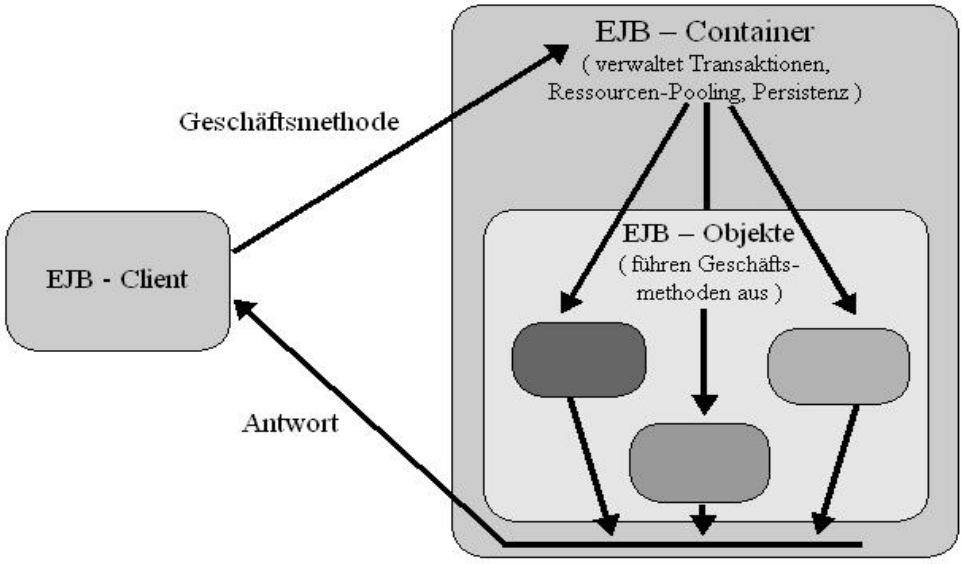
Die EJB – API ist eine standardmäßige Java – Erweiterung und ist im Package javax.ejb und dessen Sub – Packages enthalten. Diese Erweiterungs-API müssen Sie explizit installieren, um die EJB - Interfaces verwenden zu können. Die neueste Version der API kann man unter *http://java.sun.com/products/ejb/* finden. Dabei ist zu bedenken, dass EJB nur eine Spezifikation dafür ist, wie verteilte Objekte in der Java – Umgebung funktionieren sollen. Um EJB – Objekte wirklich zu erstellen und zu verwenden, müssen Sie einen EJB – fähigen Server installieren. Ein J2EE – konformer Application Server stellt diesen Dienst zur Verfügung und wird zusammen mit den Standardklassen und Interfaces der EJB – API ausgeliefert.

Die aktuell freigegebene Version der EJB-Spezifikation hat die Nummer 2.0 und diese unterstützen derzeit auch die meisten Application Server .Die 2.0 – Spezifikation wurde um signifikante Eigenschaften erweitert , wie zum Beispiel lokale Interfaces zu EJBs, ein neues Objektmodell für container-verwaltete Persistenz, Message – driven EJB. Weitere Informationen zu EJBs und den folgenden Beschreibungen finden sich unter [1], [10],[11],[12],14], [15] und [22].

3.3.1 EJB – Rollen

In der RMI – Umgebung gibt es zwei fundamentale Rollen: den Client eines Remote – Objekts sowie das Objekt selbst, das als eine Art Server oder Service – Provider agiert. Diese beiden Rollen gibt es auch in der EJB-Umgebung, bei den EJBs wird noch eine weitere zugefügt: die Rolle des *Container – Providers*. Dieser ist dafür verantwortlich, einem EJB-Objekt alle zuvor erwähnten zusätzlichen Dienste zur Verfügung zu stellen: Transaktionsverarbeitung, Objektpersistenz und Ressourcen-Pooling. Der Container der EJBs ist eine rein serverseitige Entität. Der Client braucht keinen eigenen Container, um EJB – Objekte nutzen zu können, aber ein EJB – Objekt muss einen Container haben, damit es für die Verwendung durch einen Client exportiert werden kann. In der folgenden Abbildung werden die Zusammenhänge zwischen den drei EJB – Rollen dargestellt.

Abbildung 3.5 : Zusammenhang zwischen den EJB – Rollen



3.3.2 Der EJB – Client

Ein EJB - Client benutzt entfernte EJB - Objekte, damit diese auf Daten zugreifen, Aufgaben ausführen oder generell irgend etwas für ihn erledigen. Als erstes sucht ein Client in einer EJB – Umgebung das Home – Interface des von ihm benötigten EJB – Objekts. Dieses Home – Interface ist eine Art Objekt – Factory, mit der man neue Instanzen des EJB – Typs erzeugen, bestehende Instanzen suchen (nur bei den später behandelten Entity – EJB – Objekten) und EJB – Objekte löschen kann. EJB – Home – Interfaces werden von Clients mit Hilfe von JNDI auf dieselbe Weise lokalisiert, wie auf andere J2EE-Ressourcen zugegriffen wird, so zum Beispiel auf eine JDBC – DataSource oder eine JMSConnectionFactory. Ein EJB – Server veröffentlicht das Home – Interface eines bestimmten EJB – Objekts unter einem bestimmten Namen in einem JNDI – Namensraum. Der EJB – Client muss sich mit dem JNDI – Server verbinden und das EJB – Home – Interface unter dem entsprechenden Namen suchen. Die wesentlichen Schritte, die ein Client durchführt sind :

- Er holt einen JNDI – Kontext vom EJB/J2EE – Server.
- Er sucht mit diesem Kontext das Home – Interface der zu verwendenden Bean.
- Mit Hilfe dieses Home – Interface erzeugt (oder findet) er eine Referenz auf eine EJB.
- Er ruft Methoden dieser Bean auf.

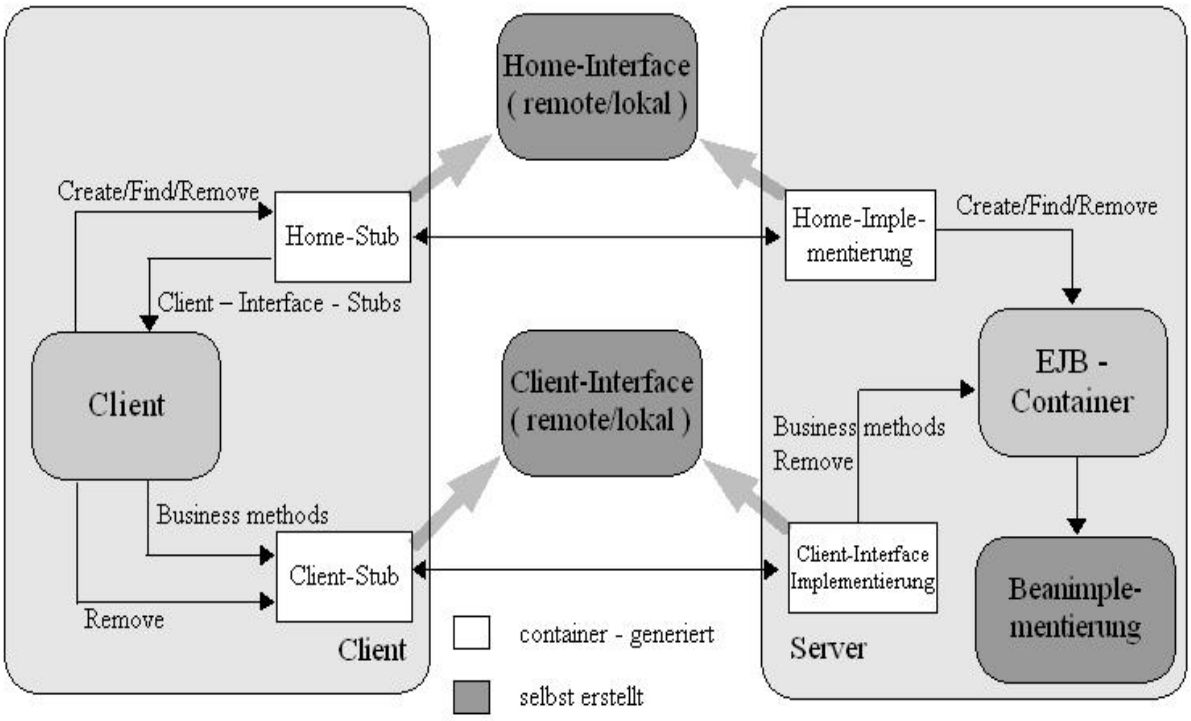
3.3.3 Der EJB – Container

Der EJB – Container implementiert die erweiterten Möglichkeiten, die EJBs über standardmäßige Remote – Objekte hinaus bieten, die mit Hilfe von RMI oder CORBA erstellt werden können. Der EJB – Container verwaltet den Lebenszyklus seiner EJB sowie die gesamte Transaktionsverarbeitung, die Sicherheit, das Ressourcen-Pooling und die Datenpersistenz. Damit erleichtert er den Client – Anwendungen und den EJB – Objekten die Arbeit und ermöglicht ihnen, sich auf ihre eigentliche Aufgabe zu konzentrieren.

Der EJB – Container stellt das Herz der EJB – Umgebung dar. Er registriert EJB – Objekte für den Zugriff durch Clients, verwaltet Transaktionen zwischen Clients und EJB – Objekten, ermöglicht Zugriffskontrollen für einzelne Methoden einer EJB und ist für die Erzeugung, das Pooling und die Vernichtung von Enterprise – Beans zuständig. Außerdem registriert der Container das Home – Interface zu jedem Bean – Typ unter einem festgelegten Namen in einem JNDI – Namensraum und ermöglicht es damit den Clients, dieses zu finden und zur Erzeugung von Enterprise – Beans zu verwenden.

Nachdem dem EJB – Container das Home - und das Client - Interface, die Implementierungsklasse sowie der Deployment Deskriptor zur Verfügung gestellt wurde, erzeugt der Container die verschiedenen Klassen, durch die diese Komponenten zusammengehalten werden. Das Home – und das Client – Interface werden vom Entwickler erstellt, während der Container, soweit erforderlich, die client – und serverseitigen Implementierungen dieser Interfaces generiert. Die folgende Abbildung zeigt die Beziehungen zwischen den containergenerierten und den vom Entwickler erstellten Klassen.

Abbildung 3.6 : Beziehungen zwischen den container – generierten und selbst erstellten Klassen



Wie der Abbildung zu entnehmen ist, können EJBs lokale und entfernte Clients unterstützen (Die EJB – 1.1 – Spezifikation unterstützt nur entfernte Clients; in EJB 2.0 wurde das Konzept der lokalen Clients wie auch deren Unterstützung eingeführt). Wenn ein entfernter Client über JNDI nach dem entfernten Home – Interface der Bean sucht, erhält er eine Instanz der entfernten Stub – Klasse. Alle bei diesem Stub aufgerufenen Methoden werden entfernt über RMI oder IIOP bei dem korrespondierenden Home – Implementierungsobjekt auf dem EJB – Server aufgerufen. Gleichermaßen erhält der Client, wenn er mit Hilfe des entfernten Home – Stub Beans erzeugt oder findet, entfernte Objekt – Stubs und bei diesen Stubs aufgerufene Methoden werden über RMI an die entsprechenden Implementierungsobjekte auf dem Server weitergeleitet. Diese entfernten Objekte sind durch den EJB – Container mit den jeweils korrespondierenden Enterprise – Bean – Objekten verknüpft, die Instanzen der Bean – Implementierungsklassen sind. Lokale Clients interagieren mit EJBs in einer vereinfachten und dadurch effizienteren Weise. Home – Objekte und Bean – Interfaces werden genauso wie die entfernten Objekte und Interfaces erworben, aber die erhaltenen Objekte sind nicht entfernte Objekte, die mit ihren Implementierungen lokal innerhalb des EJB – Containers interagieren.

Alle Client – Anfragen zum Erzeugen, Suchen und Löschen von EJBs oder zum Aufruf von EJB –

Methoden werden durch den EJB – Container vermittelt. Entweder verarbeitet er sie selbst, oder er gibt die Anfragen an die entsprechenden Methoden des EJB – Objekts weiter. Nachdem ein Client eine Referenz auf ein Interface für ein EJB - Objekt erhalten hat, schaltet sich der Container in alle Methodenaufrufe bei der Bean ein, um die Bean mit den erforderlichen Maßnahmen zur Transaktionsverwaltung und Sicherheit zu versorgen. Außerdem bietet der Container Unterstützung bei der Persistenz von Enterprise – Beans, indem er entweder den Bean – Zustand selbst speichert und lädt oder indem er die Bean informiert, wenn sie ihren Zustand persistent speichern oder wieder laden muss.

Ein Container kann während seiner Lebenszeit mehrere EJB – Objekte und Objektarten verwalten.
Dabei hat er gewisse Freiheiten bei der Verwaltung von Ressourcen zu Performance – oder anderen Zwecken. Beispielsweise kann ein Container entscheiden, ob er eine Bean vorübergehend serialisiert und auf dem Dateisystem des Servers oder auf einem anderen persistenten Speicher ablegt werden soll. Dies wird als *Passivieren* einer Bean bezeichnet. Das EJB – Objekt wird darüber informiert und erhält die Möglichkeit, gegebenenfalls gemeinsame Ressourcen oder nicht zu serialisierende, transiente Daten freizugeben. Nachdem sie wieder aktiviert worden ist, wird die Bean erneut informiert, so dass sie transiente Zustände wieder aufbauen und gemeinsame Ressourcen wieder öffnen kann.
Bei der Inbetriebnahme eines EJB – Objekts in einem EJB – Server kann man angeben, wie der Container die Bean zur Laufzeit bezüglich der Transaktionsverwaltung, des Ressourcen – Pooling, der Zugriffskontrollen und der Datenpersistenz behandeln soll. Für diesen Zweck werden Deployment Deskriptoren verwendet, die Parametereinstellungen für diese diversen Optionen enthalten. Diese Einstellungen können für jede Inbetriebnahme eines EJB – Objekts entsprechend angepaßt werden.

3.3.4 Das EJB – Objekt

Um ein EJB – Objekt für einen EJB – Container vollständig zu beschreiben, müssen ihm typischerweise drei Java – Interfaces beziehungsweise Klassen zur Verfügung gestellt werden:

- ein Bean – Interface (entfernte und/oder lokale Version)
- ein Home – Interface (entfernte und/oder lokale Version)
- eine Enterprise – Bean – Implementierung

Abhängig von der Art der entwickelten EJB kann es mehr oder weniger Klassen und Interfaces geben,

die zur Verfügung gestellt werden müssen (z.B. Primärschlüssel – Klassen für bestimmte Entity – EJBs, Message – driven Beans, die kein Home – Interface benötigen), aber die oben aufgeführten Elemente sind typisch für die meisten Session und Entity EJBs. Das Bean – Interface definiert die extern aufrufbaren Operationen der EJB. Das entfernte Interface der EJB gibt die Operationen an, die entfernte Clients bei der Bean aufrufen können, während das lokale Interface die Operationen angibt, die lokale Clients (andere EJBs, Java – Beans und Servlets, die in derselben Java Virtual Machine laufen) bei der Bean aufrufen können. Die lokalen Home – und Bean – Interfaces wurden in EJB 2.0 neu hinzugefügt, um Komponenten eine effiziente Möglichkeit für den Zugriff auf EJBs in derselben JVM zu ermöglichen. Vor EJB 2.0 konnte man auf die Bean nur über entfernte Interfaces zugreifen. Ein (lokaler oder entfernter) Client führt Methodenanfragen über einen Stub durch, der von dem Bean – Interface abgeleitet ist, und irgendwann finden diese Anfragen ihren Weg zu der korrespondierenden Bean – Instanz, die innerhalb des EJB – Containers läuft. Das Home – Interface stellt eine Bean – Factory dar und bietet dem Client die Möglichkeit, die von ihm verwendeten EJB – Objekte zu erzeugen, zu lokalisieren und zu zerstören.

Es folgt ein entferntes Client – Interface für eine einfache Account – Bean:

```
import javax.ejb.*;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Account extends EJBObject {
        // Zugriff auf Accountdaten
        public String getNumber( ) throws RemoteException;
        public void setNumber( String) throws RemoteException;
        public Double getBalance( ) throws RemoteException;
        public void setBalance( ) throws RemoteException;
}
```

Dieses Interface zeigt die für entfernte Clients zugänglichen Geschäftsmethoden. Wenn ein Client durch das entfernte Home – Interface eine Referenz auf diese Bean erhält, bekommt er in Wirklichkeit einen Stub, der dieses Interface Account implementiert.

Hier ist ein entferntes Home – Interface für die obige Account – Bean:

```
import javax.ejb.*;
import java.rmi.RemoteException;
// Das "Home"-Interface der AccountBean. Dieses Interface bietet
// Methoden zum Erzeugen von Beans auf dem Server. Für die
// Implementierung dieses Interface ist der Container-Provider
// verantwortlich, der dazu höchstwahrscheinlich automatisch
// generierte Java-Klassen verwendet, die aus dem Bytecode des
// Interface abgeleitet sind.
public interface AccountHome extends EJBHome {
        // Erzeuge einen neuen (datenlosen) Account.
        public Account create( ) throws CreateException, RemoteException;
        // Erzeuge einen Account mit Daten.
        public Account create(String accountNumber, String accountBalance)
        throws CreateException, RemoteException;
        // Suche einen Account über die Kontonummer (Primärschlüssel).
        public Account findByPrimaryKey(String accountNumber)
        throws RemoteException, FinderException;
}
```

Dieses Home – Interface enthält Methoden, mit denen man Account - Beans erzeugen kann und finden kann, wenn sie bereits auf dem Server existieren.

Die Implementierung des EJB – Objekts muss alle Geschäftsmethoden implementieren, die in dem entfernten und dem lokalen Interface offengelegt sind, und zusätzlich einige Methoden, mit denen der Container das Objekt über verschiedene Ereignisse der Lebenszeit informiert. Das EJB – Objekt muss nicht das entfernte oder lokale Interface direkt implementieren. Bei EJB sorgt der Container dafür, dass Methodenaufrufe beim Bean – Interface zum EJB – Objekt übertragen werden. Man muss allerdings sicherstellen, dass das EJB – Objekt über Methoden verfügt, die zu den Signaturen der Methoden des Client-Interfaces passen.

Die Home – und Client – Interfaces einer EJB sind ausschließlich für die Nutzung durch den Client vorgesehen. Sie ermöglichen dem Client, EJB – Objekte zu erzeugen und ihre Methoden aufzurufen.

Die Implementierungsklasse einer Bean unterstützt mit ihrer Funktionalität nicht nur die für den Client zugänglichen Methoden, sondern auch solche Methoden, die der Container benötigt, damit er das EJB – Objekt zum Beispiel über transaktions – oder persistenzbezogene Ereignisse informieren kann.

Zusätzlich zu den Interfaces, die den Typ der EJB - Komponente beschreiben, wird dem Container auch ein Deployment Deskriptor zur Verfügung gestellt. Der Deployment Deskriptor teilt dem Container mit, unter welchem Namen das Home - Interface der Bean im JNDI registriert werden soll, wie die Transaktionen für die Bean verwaltet werden, welche Zugriffsrechte entfernte Identitäten für den Zugriff auf die Methoden der EJB erhalten und wie bei der Bean mit der Persistenz umzugehen ist. Außerdem deklariert er Referenzen auf Ressourcen, die die EJB in ihrer Implementierung verwendet, und bietet eine Reihe weiterer Parameter dazu, wie die EJB vom Container verwaltet werden soll. Der Container leistet die ganze Schwerarbeit, die nötig ist, um diese Dienste zur Verfügung zu stellen, aber das EJB-Objekt muss dem Container mitteilen, wie mit diesen Diensten umgegangen werden soll.

3.3.5 Arten von EJB

Es gibt drei Grundtypen von Enterprise JavaBeans: *Session*, *Entity* und *Message-driven* Beans. Auf die letztere Variante wird in einem späteren Kapitel eingegangen. Der entscheidende Unterschied zwischen diesen Komponentenmodellen besteht darin, wie sie während ihrer Lebenszeit vom EJB – Container verwaltet werden und welche Komponentendienste ihnen durch den Container zur Verfügung gestellt werden.

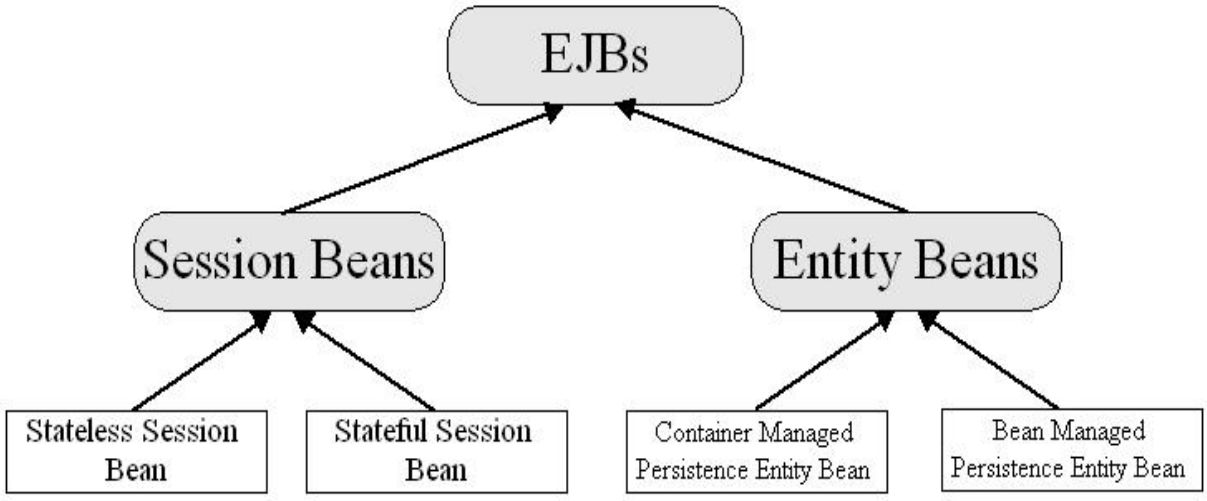


Abbildung 3.7 : EJB – Typen

3.3.5.1 Session Beans

Session Beans werden dazu verwendet, serverseitige Aufgaben im Auftrag des Clients auszuführen, Ressourcen (wie Kurse und Tabellen) zugänglich zu machen und das Zusammenspiel von Entity Beans zu organisieren. Session Beans haben keine Persistenz und repräsentieren keine Daten aus einer Datenbank (sehr wohl kann es aber zu ihren Aufgaben gehören, Daten aus der Datenbank zu lesen und zu schreiben) und enkapsulieren typischerweise die Business Logik einer Anwendung. Typische Beispiele für Session Beans sind:

- Warenkorb, der sich während einer Session aufbaut
- Währungsumrechner

Es gibt zwei grundsätzliche Arten von Session Beans: Stateless und Stateful.
Stateless Session Beans behalten keine Zustandsinformation von einem Aufruf zum nächsten. Man kann sie wie Funktionsaufrufe verstehen, die allein mit den übergebenen Parametern arbeiten. Zum Beispiel gibt man einen Wert und eine Währung an und erhält den Betrag in Euro.
Sie haben keine Persistenz in der Datenbank und repräsentieren deshalb auch keine Datenbankdaten. Vielmehr stellen sie Geschäftsprozesse oder Abläufe dar, die auf Initiative eines Clients hin ausgeführt werden, der ihre Funktionen benutzt. Jeder Methodenaufruf ist hierbei unabhängig von anderen und vorangegangenen Methodenaufrufen. Aufgrund ihrer Zustandslosigkeit sind sie für den EJB Container relativ leicht zu handhaben und verbrauchen nicht viele Ressourcen.
Stateful Session Beans behalten einen Zustand von Aufruf zu Aufruf. Sie beziehen sich immer auf einen Client und behalten sozusagen als „verlängerter Arm“ des Clients ihren Zustand. Verschiedene Clients "sharen" niemals eine Session Bean. Wenn ein Client eine Session Bean erstellt, ist sie nur diesem Client zugänglich.
Bei einem Bestellvorgang werden, die zuvor selektieren Produkte in der Bestellung belassen. Beim Hinzufügen eines Artikels ein einen Warenkorb sind die vorher hinzugefügten weiterhin vorhanden.

Der Container kann aus Container Management Gründen eine Session Bean zeitweise aus dem Memory nehmen und in einen Secondary Storage (Disk) schreiben. Dieser Transfer wird Instanz „Passivation“ genannt. Der umgekehrte Transfer ist „Activation“. Der Container sollte eine Session

Bean nur passivieren, wenn sie nicht Teil einer Transaktion ist. Damit der Container den „State“ einer Session Bean besser managen kann, wird ihm im Deployment Deskriptor der Bean mitgeteilt, ob es sich um eine stateless oder stateful Session Bean handelt.

Session Beans werden entweder durch ein Time out oder durch ein explizites Entfernen durch den Client zerstört. Nach Überschreiten eines bestimmten Zeitwerts, löscht der Container die Session Beans. Wie lange der Container bei Inaktivität wartet, bis er eine Session Bean entfernt, ist in den Container Einstellungen konfigurierbar. Ein Client muss sich darauf einstellen, das sein entferntes Interface nicht mehr gültig ist, weil der Container die Session Bean nach einer längeren Zeit von Inaktivität entfernt hat.
Ein Client kann aber auch durch Aufrufen von remove() auf das entfernte EJBObject, die Session Bean entfernen.
In jedem Fall wird die ejbRemove() der Session Bean vom Container vor der Zerstörung aufgerufen.

3.3.5.2 Entity Beans

Entity Beans repräsentieren für den Client eine objektorientierte Sicht auf einen Datensatz, z.B. eine Zeile in einer Datenbank. Sie erlauben im Gegensatz zu Session Beans auch Mehrbenutzerbetrieb, das heißt, dass auf eine Instanz eines Entity Beans gleichzeitig mehrere Benutzer zugreifen können. Der Container, in den Entity Beans während der gesamten Lebensdauer eingebettet sind, stellt dazu entsprechende Mechanismen bereit, um z.B. Sicherheit, Transaktionskonsistenz und Parallelität sicherzustellen. Für den Client ist der Container transparent, so dass keine zusätzlichen Schnittstellen existieren, die eine Manipulation des Containers ermöglichen würden. Da Entity Beans nicht an einen einzelnen Client gebunden sind, endet ihre Lebensdauer nicht nach dem Beenden einer Client – Verbindung. Im Gegensatz zu Session Beans können bzw. müssen sie sogar nach einem Systemausfall automatisch wiederhergestellt werden, da in der Regel ihre Existenz an das Vorhandensein der mit ihnen verbundenen Daten gebunden ist. Das heißt, die Erstellung einer Instanz eines Entity Beans erzeugt z.B. automatisch eine neue Zeile in einer Datenbank und fügt die bei der Erstellung mit übergebenen Daten des erstellten Datensatzes der Datenbank hinzu. Wird die Instanz entfernt, wird automatisch der mit dieser Instanz verbundene Datensatz aus der Datenbank gelöscht. Je nachdem, zu welchem Zweck Entity Beans entwickelt worden sind, unterscheidet man auch bei ihnen zwischen zwei verschiedenen Arten:

- Container managed Persistence (CMP) Entity EJB
- Bean managed Persistence (BMP) Entity EJB

Bei CMP EJBs wird die Persistenz der repräsentierten Daten von dem Container, in den das EJB eingebettet ist, garantiert. Der Entwickler braucht sich bei der Erstellung dieser Art von Entity Beans also nicht darum zu kümmern, wie die Daten des Entity Beans gesichert werden. Der Container kann z.B. zum lesenden und schreibenden Zugriff auf eine relationale Datenbank eigenständig SQL – Code generieren und ausführen. Der Entwickler muss somit nur festlegen, welche Daten er von den anzusprechenden Datenbanken im Entity Bean repräsentieren möchte und verknüpft diese dann mit der Bean – Klasse.
Bei BMP Entity Bean ist es Aufgabe des Entwicklers, sich um die Art und Weise zu kümmern, wie die Daten des Entity Beans gesichert werden. Dies mag auf den ersten Blick ein Nachteil sein, in der Praxis können aber Situationen auftauchen, z.B. bei einer Datenbankanfrage mittels Join, in denen man mit BMP EJBs durch „handoptimierte“ SQL-Statements eventuell Leistungsverbesserungen erzielen kann. Joins über mehrere Datenbanken mit mehreren Tabellen sind mit CMP EJBs gar nicht oder nur schwer realisierbar, weswegen man auch hier meist BMP EJBs verwendet. Letztendlich kann der Entwickler bei BMP Entity Beans auch das Objektmodell freier gestalten, er ist hier nicht so sehr an das Datenbankschema gebunden wie bei CMP Entity Beans.

3.3.6 Unterschiede und Gemeinsamkeiten von Session und Entity Beans

3.3.6.1 Unterschiede

Im Folgenden werden noch einmal kurz die Unterschiedungskriterien zwischen Session und Entity Beans in Tabellenform zusammengefasst.

Merkmal	Session Bean	Entity Bean
Aufgabe	Repräsentiert einen serverseitigen Dienst, der Aufgaben für einen Client ausführt	Repräsentiert ein Geschäftsobjekt, dessen Daten sich in einem dauer - haften Speicher befinden(z.B. Datenbank)
Zugriff	Ist eine private Ressource für Client. Sie steht ihm exklusiv zur Verfügung	Ist eine zentrale Ressource, die Bean – Instanz wird von mehreren Clients gleichzeitig genutzt und ihre Daten stehen allen Clients zur Verfügung
Persistenz	Nicht persistent; wird Client oder Server terminiert, ist die Bean nicht mehr verfügbar	Persistent; wird Client oder Server terminiert, befindet sich der Zustand der Bean auf einem persistenten Speicher, somit ist die Wiederher - stellung zu einem späteren Zeitpunkt möglich

Tabelle 1: Unterscheidungskriterien zwischen Session und Entity Beans

3.3.6.2 Gemeinsamkeiten

Die Gemeinsamkeiten von Session und Entity Beans bestehen vor allem in ihrem Aufbau und der Art, wie der Zugriff auf diese EJB – Objekte stattfindet. Auf beide wird über das entfernte oder lokale Client – bzw. Home – Interface zugegriffen, was auf den gleichen Aufbau zurückzuführen ist. Das heißt also, beide Arten bestehen aus dem Client – und dem Home – Interface , jeweils in der entfernten oder bei der EJB – 2.0 – Spezifikation in der lokalen Variante und der Implementierungsklasse der Bean. Sie unterliegen den selben Beschränkungen, die für alle Beans gelten. Diese sind:

- es dürfen keine statischen Variablen verwendet werden, nur statische Konstanten
- es dürfen keine Thread – Synchronisationsmechanismen verwendet werden
- es darf keine AWT – Funktionalität und kein Dateisystemzugriff genutzt werden
- es dürfen keine Klassen aus java.security, keine Sockets und Threads verwendet

werden

- es dürfen keine nativen Bibliotheken geladen und es darf niemals this übergeben werden

3.3.7 Lebenszyklus von Session und Entity Beans

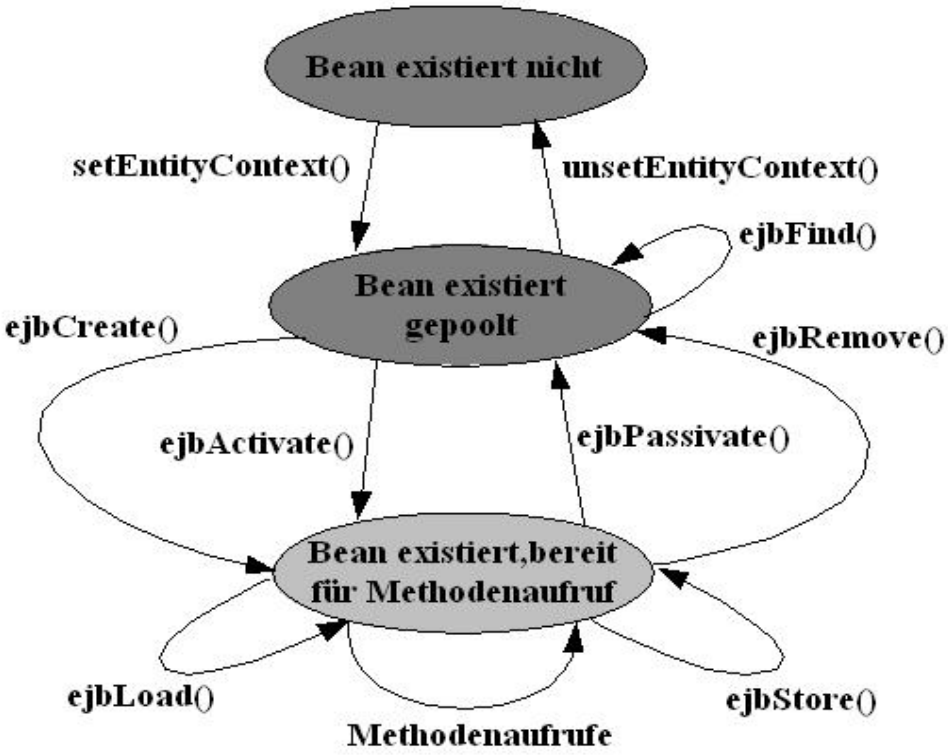
Der Lebenszyklus bezeichnet die Darstellung aller Zustände und deren Übergänge. Für Entity Beans gibt es nur einen Lebenszyklus, da die Unterscheidung des Persistenztyps nichts an den möglichen Zuständen ändert. Im Gegensatz dazu gibt es Unterschiede bei den Session Beans, denn bei der Stateful – Variante einer Session Bean wird intern ein Zustand verwaltet, was bei der Stateless – Variante nicht der Fall ist. Dadurch lassen sich Stateful Session Beans nicht in einem Pool verwalten, wenn Sie gerade nicht in Benutzung sind. Sie werden dann vom EJB – Container passiviert und bei Bedarf wieder aktiviert. Das heißt, ihr Inhalt wird serialisiert und auf einem Speichermedium abgelegt, um Ressourcen zu schonen. Bei den Entity Beans und den Stateless Session Beans werden die nicht genutzten Instanzen der Beans in einem Pool verwaltet. Damit wird verhindert, dass die Instanzen der Beans nicht ständig erzeugt und zerstört werden müssen. Dies erleichtert die Arbeit des Garbage – Collectors erheblich und vermindert auch die Anzahl der neu zu erzeugenden Objekte und spart somit Ressourcen. Im folgenden werden die Zustände der Beans kurz erläutert.

3.3.7.1 Entity Beans

Es gibt drei mögliche Zustände bei den Entity Beans:

- *Bean existiert nicht* die Bean – Instanz existiert noch nicht
- *Bean existiert, gepoolt* die Instanz existiert, ihr ist aber keine Identität zugeordnet
- *Bean existiert, bereit für Methodenaufruf* der existierenden Instanz ist eine Identität zugeordnet, die Methoden können vom Client aufgerufen werden

Abbildung 3.8 : Lebenszyklus einer Entity Bean



Um einer Beaninstanz eine Identität zuzuweisen, muss entweder eine Create – oder Findermethode des Home – Interfaces aufgerufen werden. Sind nicht genug Instanzen vorhanden, wird eine neue erzeugt und mit der Methode *setEntityContext()* die Umgebung gesetzt. Mit der Methode *unsetEntityContext()* wird die Instanz wieder aus dem Pool entfernt. Ein Client wird nach dem Aufruf im Home – Interface nur ein Client – Interface des Beans zugewiesen. Diesem Objekt wird nun durch den automatischen Aufruf der Methoden *ejbLoad()* und *ejbActivate()* eine Identität zugewiesen. Damit ist die Instanz bereit für einen Methodenaufruf. Wenn die Daten des Beans verändert werden, wird die Methode *ejbStore()* vom EJB – Container aufgerufen, um die neuen Daten persistent zu halten. Durch einen Aufruf von *ejbRemove()* werden die Daten aus der Datenbank entfernt und die Instanz wird in den gepoolten Zustand überführt, da somit die Identität nicht mehr existent ist. Wenn der Client die Bean – Instanz nicht mehr benutzt wird *ejbPassivate()* aufgerufen und auch damit wird die Bean in den gepoolten Zustand versetzt. Beim Aufruf von *ejbActivate()* und *ejbPassivate()* müssen eventuell benutzte Ressourcen, welche nicht persistent gehalten werden, neu initialisiert bzw. wieder freigegeben werden. Tritt ein Systemfehler auf, wird die gerade benutzte Instanz entfernt. Das bedeutet, es wird unsetEntityContext() aufgerufen und die Bean wird in den Zustand *Bean existiert nicht* versetzt.

3.3.7.2 Stateless Session Bean

Das Stateless Session Bean wird ähnlich verwaltet wie das Entity Bean. Allerdings gibt es da nur zwei mögliche Zustände:

- *Bean existiert nicht* die Instanz existiert nicht
- *Bean existiert, gepoolt, bereit für Methodenaufruf* die Instanz befindet sich im Pool und ist bereit für Methodenaufrufe

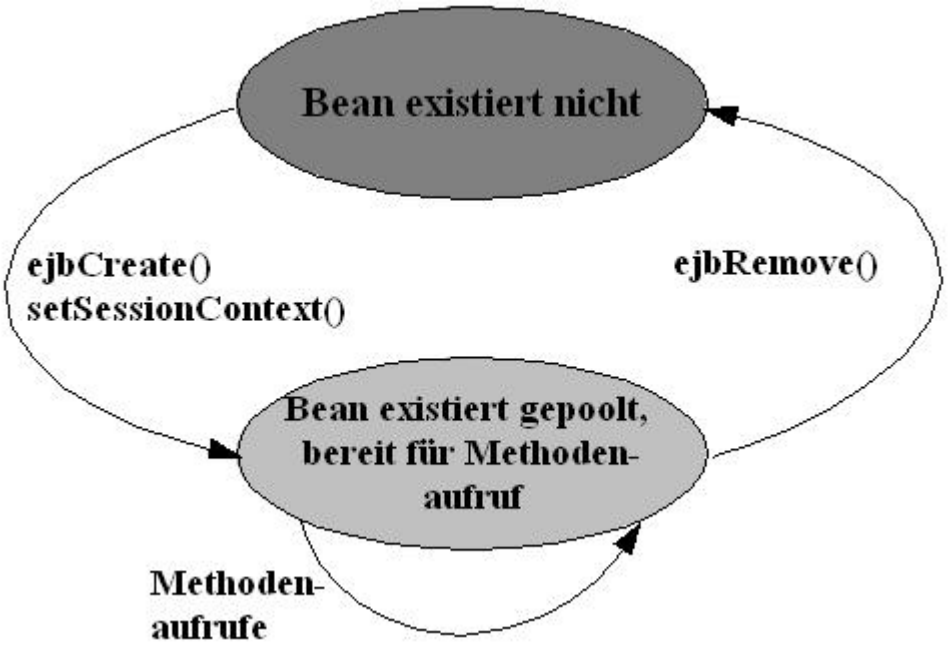


Abbildung 3.9 : Lebenszyklus einer Stateless Session Bean

Ein Methodenaufruf bewirkt, dass eine Instanz des Beantyps aus dem Pool entnommen, diese Methode ausgeführt und danach wieder in den Pool gegeben wird. Dies ist möglich, da Session Beans in der Stateless – Variante keine Werte zwischen den Methodenaufrufen speichern. Dadurch ist es egal, mit welcher Instanz die Methode ausgeführt wird. Ein Systemfehler oder ein Aufruf der Methode *ejbRemove()* führen dazu, dass die Instanz aus dem Pool entfernt wird.

3.3.7.3 Stateful Session Bean

Da beim Stateful Session Bean zwischen zwei Methodenaufrufen der interne Zustand des Beans verändert werden kann, ist das Verfahren des Poolings hier nicht anwendbar. Der Zustand des Beans muss gespeichert werden, auch wenn das Bean zur Zeit nicht benutzt wird. Daraus ergeben sich vier Zustände:

- *Bean existiert* die Instanz existiert nicht
- *Bean exitiert, bereit für Methodenaufruf* die Instanz existiert und ist bereit für Methodenaufruf
- *Bean passiviert* die Instanz wurde passiviert
- *Bean bereit in Transaktion* die Instanz befindet sich innerhalb einer Transaktion und ist bereit für Methodenaufrufe

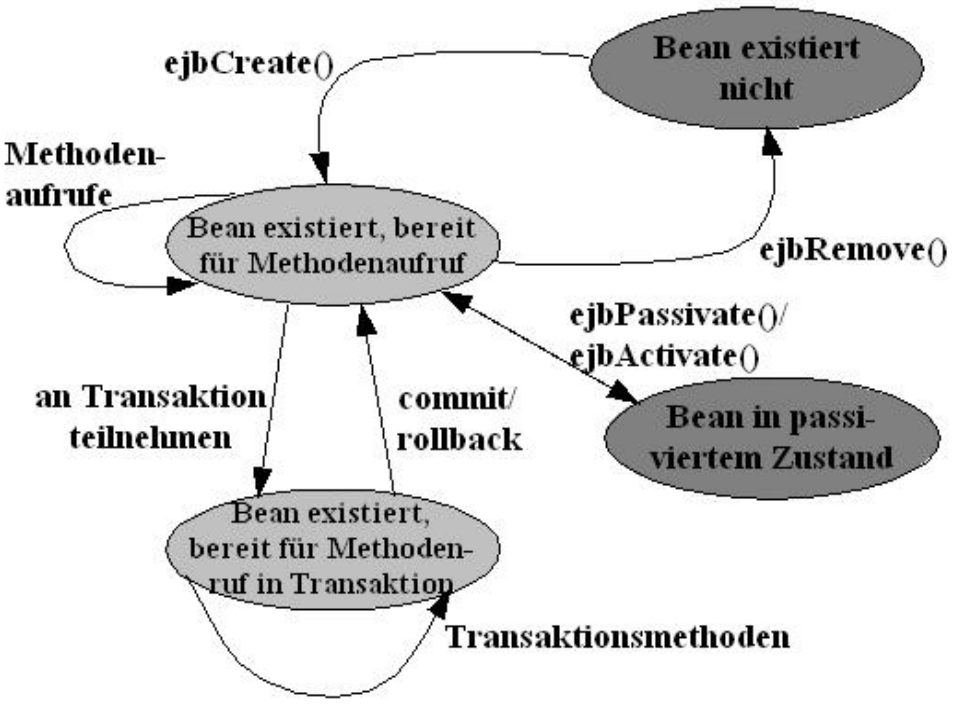


Abbildung 3.10 : Lebenszyklus einer Stateful Session Bean

Wenn eine neue Bean – Instanz benötigt wird, erstellt der Container diese, wie vorher auch, mit *ejbCreate()* und *setSessionContext().* Sie ist nun im Zustand *Bean existiert, bereit für Methodenaufruf* und wartet auf Methodenaufrufe. Sollen diese Methoden innerhalb einer Transaktion aufgerufen werden, wird eine Transaktion erstellt und das Bean befindet sich im Zustand *Bean bereit in Transaktion.* Die Methoden werden ausgeführt und die Transaktion mit Commit oder Rollback beendet. Nun befindet sich das Bean wieder im vorangegangenen

Zustand *Bean existiert, bereit für Methodenaufruf.* Damit der EJB – Container Ressourcen spart, werden nicht alle Instanzen im Speicher gehalten, sondern eine zur Zeit nicht genutzte Bean wird durch den Aufruf von
ejbPassivate() in den Zustand *Bean passiviert* überführt. Durch *ejbActivate()* wird sie wieder in den Zustand versetzt, in dem sie bereit für Methodenaufrufe ist. Eine Bean darf nicht passiviert werden, wenn sie im Zustand *Bean bereit in Transaktion* ist, da erst die Transaktion geschlossen werden muss. Bei einem Systemfehler oder durch den Aufruf der Methode ejbRemove() wird die Bean – Instanz aus dem Speicher entfernt.

3.3.8 Der Deployment Deskriptor

Der Deployment Deskriptor ist eine Datei im XML – Format, die eine oder mehrere Beans, deren Zusammenwirken und die Art, wie der EJB – Container sie zur Laufzeit behandeln soll, beschreibt. Er enthält hauptsächlich deklarative Informationen, welche nicht im Bean – Code zu finden sind. Dies sind vor allem Informationen über die Struktur der Bean und ihrer Abhängigkeiten zu anderen Beans oder Ressourcen wie z. B. einer Datenbankverbindung. Ausserdem können im Deployment Deskriptor Umgebungsvariablen gesetzt werden, die vom Bean ausgelesen werden können und somit ihr Verhalten beeinflussen. Dies führt zu einer höheren Flexibilität, da das selbe Bean in verschiedenen Umgebungen eingesetzt werden kann und nur der Deployment Deskriptor angepasst werden muss.
Anhand von Ausschnitten des Deployment Deskriptors für die WOMBank – Anwendung werden nun kurz die wichtigsten Attribute erläutert.

- *ejb-jar* der Anfang des Deskriptors, es beinhaltet die Beschreibungen für alle Beans, die im selben jar-File wie der Deskriptor liegen
- *enterprise-beans* beinhaltet die Beschreibungen der einzelnen Beans
- *entity* beschreibt ein einzelnes Entity Bean
- *session* beschreibt ein einzelnes Session Bean
- *session-type* legt die Art des Session Beans fest: *stateless* oder *stateful*
- *description* liefert eine Beschreibung des jeweiligen Kontextes, diese ist nur zur verbesserten Lesbarkeit für andere Nutzer
- *ejb-name* gibt den Namen der Bean an, über den die Klasse zum Beispiel mittels JNDI angesprochen werden kann
- da auf eine Enterprise – Bean nur über Interfaces zugegriffen werden kann, muss der EJB – Container deren Klassen und die Bean – Klasse selbst kennen

- ➔ *home* gibt die Klasse des Home – Interfaces mit vollständiger Packagestruktur an
- ➔ *remote* gibt die Klasse des entfernten Client – Interfaces mit vollständiger Packagestruktur an
- ➔ *ejb-class* gibt die Bean – Implementierungsklasse des Enterprise Beans mit vollständiger Packagestruktur an

- *primary-key-class* gibt die Klasse des Primary – Keys mit vollständiger Packagestruktur an, dieses Attribut kommt selbstverständlich nur bei Entity Beans zum Einsatz
- *cmp-field* gibt mit Unterattribut field-name die Attribute einer Klasse an, die vom Container automatisch persistent gehalten werden sollen
- *primary-key-field* gibt das entsprechende Primärschlüsselattribut der Beanklasse an
- *method* legt eine Methode fest, das Attribut besteht aus:
  - ➔ *ejb-name* der Name der Bean
  - ➔ *method-name* der Name der Methode
- *ressource-ref* legt den Zugriff auf externe Ressourcen fest:
  - ➔ *description* eine Beschreibung der Ressource
  - ➔ *res-ref-name* der Name der Ressource (meist eine Datenbank)
  - ➔ *res-type* der Typ der Ressource (meist *javax.sql.DataSource*)
  - ➔ *res-auth* die Sicherheitsstrategie beim Zugriff, entweder Container oder Application
- *env-entry* legt eine Umgebungsvariable fest, welche von der Beans ausgelesen werden kann und so deren Verhalten an die gewünschte Umgebung angepasst
  - ➔ *description* eine Beschreibung der Variablen
  - ➔ *env-entry-name* der Name der Variablen, kann mit JNDI durch java:comp/env/Name gefunden werden
  - ➔ *env-entry-type* nur einfache Datentypen wie String, Integer oder Double sind erlaubt
  - ➔ *env-entry-value* der Wert der Variablen
- *transaction-type* gibt an, wer sich um die Transaktion kümmert, der Container oder Application
- *assembly-descriptor* der Application-Assembler wird umgeleitet, er legt das Verhalten der Bean fest, dazu gehören Transaktionssteuerung, das Verteilen von Sicherheitsrollen und das Vergeben von Zugriffsrechten auf einzelne Klassen oder Methoden
- *security-role* mit Unterattribut role-name, legt eine Rolle, mit deren Hilfe bestimmte Zugriffsrechte gegeben oder verweigert werden können
- *container-transaction* legt die Methoden fest, für die ein Transaktionsattribut gesetzt

werden soll

- *trans-attribute* setzt das entsprechende Transaktionsattribut, es gibt:
  - ➔ *NotSupported* diese Methode unterstützt keine Transaktion, sie wird nicht in einer Transaktion ausgeführt. Deshalb darf im Code nicht versucht werden, ein Rollback durchzuführen.
  - ➔ *Required* diese Methode wird immer in einer Transaktion ausgeführt. Wenn sie nicht mit einer globalen Transaktion aufgerufen wird, startet der EJB – Container eine neue Transaktion
  - ➔ *Supported* diese Methode wird mit oder ohne Transaktion ausgeführt. Wird sie in einer Transaktion aufgerufen, wird sie in dieser ausgeführt, wenn nicht, wird sie ohne Transaktion abgearbeitet
  - ➔ *RequiresNew* für diese Methode wird immer eine neue Transaktion gestartet, in der sie abgearbeitet wird
  - ➔ *Mandatory* diese Methode muss immer in einer Transaktion aufgerufen werden. Ansonsten wird eine TransactionRequiredException geworfen
  - ➔ *Never* diese Methode darf nicht in einer Transaktion aufgerufen werden. Dies würde zu einer RemoteException führen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN" "http://java.sun.com/dtds/ejb-jar_1_1.dtd">
  <ejb-jar id="ejb-jar_ID">
    <description>The WOMBank CMP EJB</description>
    <display-name>WOMBank EJBs</display-name>
    <enterprise-beans>
      ...
        <entity id="WBBankAccount">
        <ejb-name>WBBankAccount</ejb-name>
        <home>ycubas.ejb.entity.wombank.bankaccount.WBBankAccountHome</home>
        <remote>ycubas.ejb.entity.wombank.bankaccount.WBBankAccount</remote>
        <ejb-class>ycubas.ejb.entity.wombank.bankaccount.WBBankAccountBean</ejb-
class>
        <persistence-type>Container</persistence-type>
```

```
        <prim-key-
class>ycubas.ejb.entity.wombank.bankaccount.WBBankAccountPK</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-field id="WBBankAccount_username">
          <field-name>username</field-name>
        </cmp-field>
        <cmp-field id="WBBankAccount_type">
          <field-name>type</field-name>
        </cmp-field>
          <cmp-field id="WBBankAccount_number">
          <field-name>number</field-name>
        </cmp-field>
        <cmp-field id="WBBankAccount_balance">
          <field-name>balance</field-name>
        </cmp-field>
        <resource-ref >
          <description>resource ref is used by the ws390 runtime in support of CMP .
</description>
          <res-ref-name>ws390rt/cmp/jdbc/CMPDS</res-ref-name>
          <res-type>javax.sql.DataSource</res-type>
          <res-auth>Container</res-auth>
        </resource-ref>
      </entity>
        ...
      <session id="HandleLogin">
        <description>WOMBank SessionBean to handle login</description>
        <display-name>WOMBank Handle Login Stateless Session</display-name>
        <ejb-name>HandleLogin</ejb-name>
        <home>ycubas.ejb.session.wombank.login.HandleLoginHome</home>
        <remote>ycubas.ejb.session.wombank.login.HandleLogin</remote>
        <ejb-class>ycubas.ejb.session.wombank.login.HandleLoginBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
```

```
      ...
  </enterprise-beans>

  <assembly-descriptor id="AssemblyDescriptor_ID">
    <container-transaction>
      <method>
        <ejb-name>WBPassWd</ejb-name>
        <method-name>*</method-name>
      </method>
      ...
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>HandleLogin</ejb-name>
        <method-name>*</method-name>
      </method>
        ...
      <trans-attribute>Never</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

3.3.9 Die EJB – 2.0 – Spezifikation

Am 22. August 2001 wurde die EJB – 2.0 – Spezifikation im Status "Final Release" veröffentlicht. Die Gebiete, auf denen schwerpunktmäßig Änderungen oder Neuerungen gegenüber früheren Spezifikationen aufgetreten sind, werden im Folgenden kurz erläutert :

Java Message Service (JMS)

Die JMS Spezifikation zur asynchronen Kommunikation wird in der EJB™ 2.0 Spezifikation benutzt, um das Konzept der Message Driven Beans zu ermöglichen. Die JMS API ist von Sun und einigen Partnerunternehmen entwickelt worden, um es Anwendungen zu ermöglichen, Nachrichten zu er-

stellen, zu senden, zu empfangen und zu lesen. Die JMS – Kommunikation besitzt die Eigenschaften asynchron und reliable.
Asynchron bedeutet, dass der JMS – Server Nachrichten an den betreffenden Client ausliefert, sowie sie eintreffen. Der Client muss die Nachrichten nicht anfordern.
Reliable bedeutet, dass JMS in der Lage ist, eine Nachricht sicher einmal und zwar genau einmal auszuliefern. Es besteht jedoch die Möglichkeit, auch geringere Anforderungen an die Übertragung zu stellen, wenn weniger benötigt wird (etwa Duplikate oder Nachrichtenverluste).
Es geht bei JMS nicht um die Kommunikation zwischen Nutzern, sondern vielmehr die Kommunikation zwischen Prozessen, also zwischen Anwendungen und/oder Komponenten.
Der *Producer* ist im Zusammenhang von JMS der Hersteller und Absender einer Nachricht, der *Consumer* der Empfänger der Nachricht.
Bei der Übermittlung kann man aus zwei verschiedenen Methoden wählen, P2P und Publish / Subscribe. P2P steht für Point To Point. Anders als man erwarten könnte, handelt es sich um keine über mehrere Nachrichten feststehende Verbindung zwischen einem Sender und genau einem Empfänger. Vielmehr handelt es sich bei P2P um die Verwendung sogenannter *Queues*. Eine Queue nimmt Nachrichten auf, die der einzige Sender schickt und speichert sie, und zwar solange bis ein möglicher Empfänger die Nachricht erhalten hat. Es kann also mehrere potentielle Empfänger geben. Der Name P2P rührt nun daher, dass nur einer von ihnen für das Empfangen der Nachricht ausgewählt wird. Die anderen erhalten die Nachricht nicht. Der Empfänger schickt eine Bestätigung. Bevor nicht diese Bestätigung nach möglicherweise mehreren Zustellversuchen eingetroffen ist, wird die Nachricht nicht aus der Queue gelöscht.
Publish / Subscribe (auch kurz: pub/sub) ist dagegen ein Konzept, bei dem jeder potentielle Empfänger die Nachricht erhält. Hierbei spricht man aber nicht von einer Queue, sondern von einem *Topic*. An einem Topic kann sich ein neuer Consumer jederzeit anmelden, was mit Subscribe gemeint ist. Jeder Consumer, der sich bei einem Topic eingetragen hat, erhält jede Nachricht, die der Sender schickt. Der Sender veröffentlicht quasi seine Nachricht (publish). Das hat zur Folge, dass auch kein Consumer zu einem Zeitpunkt eingetragen sein kann. pub/sub sollte also eingesetzt werden, wenn eine Nachricht keinen Abnehmer haben muss.

Message Driven Beans (MDB)

Nach Session und Entity Bean wird die Message Driven Bean als ein zusätzlicher dritter Beantyp eingeführt. Er ermöglicht asynchrone Methodenaufrufen durch Nachrichtenübermittlung via Java Message Service (JMS).

Eine Message Driven Bean ist eine zustandslose Komponente, die von Java Messages (javax.jms.Message) gesteuert wird. Wie schon die vorher spezifizierten Session und Entity Beans sind auch MDB serverseitig und transaktionssicher.
Eine MDB wird vom EJB – Container, in dem sie sich befindet, immer dann aufgerufen, wenn dieser eine Nachricht aus einer JMS Queue oder einem JMS Topic erhalten hat. Damit erfüllt eine MDB die Aufgabe eines Message Listeners.
Es ist für in einem Applicationserver befindliche MDB von zweitrangiger Bedeutung, wer Absender einer Nachricht ist. Das kann ein Java Client, eine Enterprise Bean, eine JSP – Komponente oder aber auch eine Anwendung, die selbst nicht auf J2EE aufbaut, sein. Der allgemeine Client sendet die Nachricht zum EJB – Server, ohne sich der einzelnen vorhandenen MDB im Container bewusst zu sein. Der Grund für die mangelnde Sicht auf einzelne MDB ist das Fehlen von Home und Remote Interfaces bei diesem Beantyp.

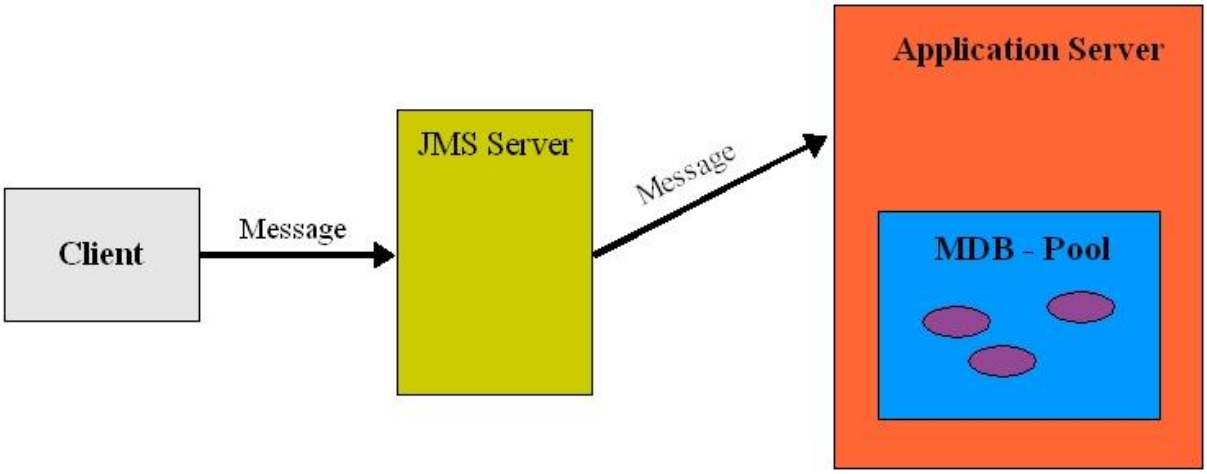


Abbildung 3.11 : Asynchroner Methodenaufruf bei Message Driven Beans

Eine MDB ist vom Typ her eine zustandslose Session Bean. Das bedeutet, dass man es mit relativ kurzlebigen Instanzen der MDB zu tun hat, die sich keinen Client merken können.
Durch die Einführung der MDB wird der EJB – Container der Version 2.0 zu einem Listener für asynchrone Aufrufe und ruft direkt (ohne Interfaces) die MDB auf, die sich dann wie eine zustandslose Session Bean verhält.
Alle Instanzen eines MDB – Typs sind gleich, da sie nicht direkt für den Client sichtbar sind und keine Zustände annehmen. Daraus resultiert die Möglichkeit für den Container, Pools aus MDB – Instanzen zu bilden, um Skalierbarkeit zu erzeugen.

Local Interfaces

Um Serialisierung bei Methodenaufrufen zwischen in derselben JVM befindlichen Beans zu vermeiden, ist das Local Interface eingeführt worden.

In der Enterprise JavaBeans Spezifikation Version 1.0 beziehungsweise Version 1.1 stehen ausschließlich Remote Interfaces (entfernte Client – Interfaces) zur Verfügung. Dies sind Schnittstellen deren Methoden durch Fernaufrufe über RMI (Remote Method Invocation) angesprochen werden. Auf Beans wird somit immer zugegriffen, als seien sie auf einem entfernten System gelegen. Dies hat zur Folge, dass Methodenaufrufe stets über zwischengeschaltete Stubs bzw. Skeletons laufen und dabei sämtliche Parameter serialisiert und anschließend wieder deserialisiert werden müssen. Unter diesem nicht unerheblichen Verwaltungsaufwand leidet die Performance des EJB – Ansatzes ganz erheblich.

Gerade bei Beans, die innerhalb der gleichen JVM (Java Virtual Machine) ablaufen, ist diese Vorgehensweise um so ärgerlicher, da der Performanceverlust vermeidbar erscheint. Mit der Einführung von Local Interfaces (lokalen Client – Interfaces) in der Enterprise JavaBeans Spezifikation 2.0 wird versucht, genau dieses Problem zu lösen. Es ist möglich, Beans, die in der selben JVM laufen, Aufrufparameter per Referenz übergeben. Ein solcher lokaler Methodenaufruf ist wesentlich schneller als ein entfernter Methodenaufruf.

Es ist nun zu beachten, dass die Implementierung einer Bean mit Local Interfaces (also Local Home und Local Client Interface) eine gleichzeitige Implementierung der Remote Interfaces (also Remote Home und Remote Client Interface) nicht ausschließt. Daraus ergibt sich, dass eine Bean alle vier der oben genannten Interfaces besitzen kann. Ein daraus resultierender Nebeneffekt ist die Aufgabe der Ortsunabhängigkeit (location transparency), da der Programmierer nun gezwungen ist zu entscheiden, ob der Zugriff innerhalb der gleichen JVM oder auf ein entferntes System stattfindet.

Container Managed Persistence (CMP)

Das Konzept der Container Managed Persistence (CMP) besteht seit der Version 1.1 der Enterprise JavaBeans Spezifikation. Allerdings räumt erst die Version 2.0 mit den Limitationen der voran-gegangenen Version auf und bringt eine umfangreiche Erweiterung der CMP – Architektur mit sich.

Das Ziel ist, dass der Entwickler das Datenbankschema nicht kennen muss, sondern dass durch CMP ein Mapping zwischen dem abstrakten und dem physischen Schema vollzogen wird. Die Steuerung der Persistenz soll durch den Server erfolgen. Vom Beanentwickler müssen lediglich die Objekteigenschaften angegeben werden, die gesichert werden sollen.

Um Entity Beans persistent zu machen, sieht dieses Konzept die Deklaration sämtlicher Attribute im

Deployment Deskriptor vor. Des Weiteren muss das objektrelationale Mapping, also die Abbildung zwischen den Attributen und den Datenbankstrukturen, im Deployment Deskriptor angegeben werden.
Der Server ist somit in der Lage, selbständig die benötigte Datenbanktabelle zu erzeugen bzw. Eintragungen innerhalb dieser vorzunehmen. Auch bei Änderungen von Attributen einer Entity Bean werden die Daten vom Server automatisch in der Tabelle gespeichert. Bei einer Suchanfrage kommt ebenfalls der Server zum Einsatz. Damit diese "Automatik" funktioniert, muss jeder Server ein Generatorwerkzeug mitbringen, welches in der Lage ist, aus dem Deployment Deskriptor Code zu erzeugen, der die genannte Funktionalität bietet.
Der Beanentwickler muss also keine einzige Zeile Programmcode verfassen, die mit dem Datenbankzugriff zusammenhängt. Dies ist ein klarer Vorteil und reduziert den Entwicklungsaufwand beträchtlich. Außerdem ergibt sich aus der Abhängigkeit von CMP Entity Beans von einem Generator ein weiterer Vorteil. Man ist unabhängig von der darunter liegenden Persistenzschicht, was zu einer sehr hohen Portierbarkeit führt.
Eine verbesserte Wartbarkeit ist dadurch gegeben, dass Änderungen am Namensschema bzw. an den Attributen einer Bean konsistent durch den Generator berücksichtigt werden. Auch die Fehleranfälligkeit ist vergleichsweise gering, woraus folgt, dass der generierte Code nicht so intensiv getestet werden muss, wie selbst geschriebener Quelltext.
Dies bringt jedoch den Nachteil mit sich, dass die Datenbankzugriffe nicht immer so effizient sein können, wie bei selbst geschriebenem Code. Im folgenden soll auf drei wichtige Konzepte, die im Zusammenhang mit CMP Entity Beans auftreten, näher eingegangen werden:

- Container Managed Relations (CMR)
- Cascaded Delete
- EJB – QL

Container Managed Relations (CMR)
Die Verwaltung der Relationen zwischen verschiedenen Enterprise JavaBeans kann seit der Version 2.0 der Spezifikation dem Container überlassen werden. Eine Voraussetzung dafür ist jedoch, dass es sich bei den zu verwaltenden Beans um CMP Entity Beans handelt.
Des weiteren müssen die Relationen im Deployment Deskriptor angegeben werden. In der entsprechenden Bean müssen abstrakte get() und set() Methoden definiert werden, welche den Attributen der Relationen entsprechen. Die tatsächliche Implementierung dieser Methoden erfolgt durch den

Containerhersteller, welcher wiederum einen Generator zur Verfügung stellen muss.
Sämtliche aus dem Datenbankentwurf bekannten Kardinalitäten werden unterstützt. 1-zu-1 , 1-zu-m und m-zu-n Beziehungen können im Deployment Deskriptor angegeben werden. Dabei ist es möglich, die Relationen entweder nur in eine Richtung (unidirektional) bzw. in beide Richtungen (bidirektional) zu traversieren. Die dabei verwendete Semantik ist in eindeutiger Weise in der Spezifikation 2.0 beschrieben. Eine weitere Aufgabe, die in diesem Zusammenhang dem Container zukommt, ist die Gewährleistung der referentiellen Integrität in der Datenbasis.

Cascaded Delete

Der sogenannte Cascaded Delete Begriff tritt im Zusammenhang mit Datenbanken auf. Dabei geht es um das automatische Entfernen von Datensätzen, die von einem übergeordneten Datensatz abhängig sind, sobald der übergeordnete Datensatz gelöscht wird.
Ein Beispiel dafür ist eine Rechnung, welche aus mehreren Positionen besteht. Wenn die Rechnung gelöscht wird, dann sollten auch die zugehörigen Rechnungspositionen gelöscht werden, damit die Datenbasis in einem konsistenten Zustand verbleibt. Diese Aufgabe kann dem Programmierer nun bei den Cascaded Deletes abgenommen werden und wird vom Containerhersteller übernommen.
Allerdings lassen sich Cascaded Deletes nur auf 1-zu-1 bzw. 1-zu-n Beziehungen definieren bzw. anwenden, da das Element, von dem der Löschvorgang gestartet wird, eine Kardinalität von 1 aufweisen muss. Auch hier werden die für Cascaded Deletes notwendigen Informationen im Deployment Deskriptor hinterlegt.

Enterprise JavaBeans – Query – Language

Die Abkürzung EJB – QL steht für Enterprise JavaBeans Query Language. Dabei gibt es nicht nur in der Namensgebung Ähnlichkeiten zu der Structured Query Language (SQL). Bei EJB – QL handelt es sich um eine Untermenge von SQL 92. Allerdings ist die EJB – QL eine reine Abfragesprache, was bedeutet, dass mit EJB – QL keinerlei Datenmanipulationen vorgenommen werden können. Die Enterprise JavaBeans Spezifikation 2.0 beschreibt die EJB – QL als "query specification language", welche auf dem in dem Deployment Deskriptor angegebenen abstrakten Schema der Bean arbeitet. Das heißt, dass die EJB – QL ausschließlich im Zusammenhang mit dem Deployment Deskriptor verwendet wird, sie wird nicht innerhalb der Beans implementiert.
Auf der Bean – Seite müssen lediglich die Signaturen von sogenannten findBy...()- bzw. selectBy...() - Methoden definiert werden. Die Implementierung erfolgt automatisch durch einen weiteren Generator, welcher abermals vom Containerhersteller zur Verfügung gestellt werden muss.

Run-As Security

Über ein Rollenschema wird es dem Applicationserver ermöglicht, Berechtigungen zur Ausführung von Methoden zu überprüfen. Das Run-As Security Konzept erweitert das bereits bestehende Berechtigungskonzept für Enterprise JavaBeans.

4 Archivstruktur von Webanwendungen

4.1 Java ARchive – JAR

Eine *ejb – jar* – Datei ist das standardmäßige Verpackungsformat für Enterprise JavaBeans. Dabei handelt es sich um eine normale **J**ava – **AR**chivdatei (JAR), die mit Hilfe des Hilfsprogramms *jar* erzeugt werden kann. Sie enthält aber spezielle Dateien, die alle jene Informationen zur Verfügung stellen, die ein EJB – Container zur Inbetriebnahme der in der JAR – Datei enthaltenen Beans benötigt.

In einer *ejb – jar* – Datei gibt es zwei Arten von Inhalten:

- Die Klassendateien aller Beans einschließlich ihrer Home – und Client – Interfaces sowie die Bean – Implementationen. Darüber hinaus können auch containergenerierte Klassen enthalten sein. (beispielsweise konkrete Implementierungen der Home – und Client – Interfaces).
- die Deployment Deskriptor – Dateien, wie in Abschnitt 4.3.8 beschrieben. Als Minimum muss eine *ejb – jar* – Datei, eine standardmäßige *ejb – jar.xml* – Datei enthalten sein. Verschiedene EJB – Container – Hersteller verlangen möglicherweise zusätzliche Deployment Deskriptor – Dateien, mit denen Sie produktspezifische Aspekte der Laufzeitverwaltung Ihrer EJBs einstellen können.

Die kompilierten Java – Klassen, aus denen die EJBs bestehen, befinden sich in packagespezifischen Verzeichnissen innerhalb der *ejb – jar* – Datei, genau wie es bei normalen JAR – Dateien der Fall ist. Der Deployment Deskriptor, mit dem die EJBs beschrieben und konfiguriert werden, muss in der *ejb – jar* – Archivdatei unter *META-INF/ejb-jar.xml* zu finden sein.

Manche Hersteller von EJB – Containern liefern ein Hilfsprogramm mit, das die Erzeugung von *ejb – jar* – Dateien aus Ihren Bean – Klassen erleichtert. So ist zum Beispiel im Falle des WebSphere Application Servers von IBM das Application Assembly Tool enthalten, welches das Zusammenstellen von EAR – Dateien auf einfache und verständliche Weise ermöglicht.

Bevor die Java – Archive deployed werden können, generiert der EJB – Container beziehungsweise ein Deployment – Werkzeug die Hilfs – und Wrapperklassen „um die EJB“ herum, die die Dienste zur Unterstützung von Kommunikation, Transaktionsverhalten, Sicherheitsmechanismen und

Datenbankzugriffen so realisieren, wie es in den Deployment Deskriptoren definiert worden ist.

4.2 Web ARchive – WAR

Eine WAR – Datei (**W**eb **AR**chiv) dient standardmäßig dazu, Web – Anwendungen zu verpacken.

In einer WAR – Datei gibt es drei Arten von Inhalten:

- Die Klassendateien aller Servlets, die Bestandteil der Webanwendung sind. Diese sind unter dem Verzeichnis WEB-INF/classes in einer package – spezifischen Verzeichnisstruktur zu finden.
- Die statischen HTML – Seiten, welche für das Ausführen der Webanwendung nötig sind. Es ist aber auch nötig JSP – Seiten, als dynamische Generierung von Response – Seiten zu verwenden. Diese Dokumente befinden sich innerhalb der WAR – Datei in einer Verzeichnisstruktur. Zum Beispiel könnten alle Bilder im Verzeichnis *images* abgelegt sein, während die *index.html* im context – root des Archivs liegt.
- Die Deployment Deskriptor – Dateien. Als Minimum muss eine standardmäßige web*.xml* – Datei enthalten sein, welche im Verzeichnis WEB-INF zu finden ist. Wie bei den Java Archiven ist es auch hier möglich, das noch weitere Deskriptoren, je nach Art des Produktes und des Herstellers, in dem Web – Archiv Verwendung finden.

Da in den vorangegangenen Abschnitten noch nicht auf den Deskriptor einer WAR – Datei eingegangen wurde, soll im nächsten kurz darauf eingegangen werden.

4.2.1 web.xml

Der Deployment Deskriptor eines Web – Archives ist, wie auch schon der ejb – jar – Deskriptor, eine XML – Datei, welche eine Beschreibung der Anwendung, die Namen und die Parameter der Servlets zur Laufzeit, sowie sessionrelevante Einstellungen enthält. Außerdem werden auch Sicherheitsaspekte, wie zum Beispiel der Loginmechanismus einer Anwendung definiert.

Anhand von Ausschnitten wird nun kurz auf die, für die WOMBank – Anwendung relevanten, Abschnitte eingegangen.

- *web-app* der Anfang des Deployment Deskriptors, der Tag beinhaltet alle für die Webanwendung

relevanten Informationen

- *display-name* hier kann ein Name für die Anwendung vergeben werden
- *servlet* die Konfiguration eines Servlets, enthält den Namen des Servlets, den Pfad der Klassendatei und auch eventuelle Anfangskonfigurationen beziehungsweise Parameter, die dem Servlet bei der Initialisierung übergeben werden sollen
  - ➔ *servlet-name* Name des Servlets
  - ➔ *servlet-class* der Klassenpfad des Servlets sowie der Klasse selbst
- *servlet-mapping* Einstellungen, die das spätere Aufrufen des Servlets betreffen
  - ➔ *url-pattern* Angabe, unter welcher URL das Servlet später zu erreichen ist
- *welcome-file-list* enthält eine Liste von Default – Dateien, die beim Aufruf der Anwendung angezeigt werden sollen
  - ➔ *welcome-file* Angabe der Datei, welche als Standarddatei geladen werden soll
- *login-config* Konfiguration des Loginmechnismus
  - ➔ *auth-method* Angabe der Authentifizierungsmethode

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
 <web-app>
   <display-name>WOMBank Web Application</display-name>
   ...
   <servlet>
    <servlet-name>WOMBankServlet</servlet-name>
    <servlet-class>ycubas.servlet.WOMBankServlet</servlet-class>
   </servlet>
   ...
   <servlet-mapping>
    <servlet-name>WOMBankServlet</servlet-name>
    <url-pattern>/WOMBankServlet</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
    <welcome-file>index.html</welcome-file>
   </welcome-file-list>
   <login-config>
```

```
    <auth-method>BASIC</auth-method>
  </login-config>
</web-app>
```

4.3 EAR – Archive

Eine J2EE Anwendung besteht aus Web – und EJB – Komponenten. Beide Arten von Komponenten sind, wie in den vorhergehenden Ausführungen beschrieben, in Archiven verpackt, Verhalten und Eigenschaften sind in Deployment Deskriptoren (DD) definiert. Alle Komponenten einer Anwendung sind in einem EAR (**E**nterprise **A**pplication **R**essource) zusammengestellt. Eine vollständige Anwendung kann durch Installation eines EAR – Files auf einen anderen Application Server verteilt werden und sollte sich dort wie erforderlich konfigurieren lassen.

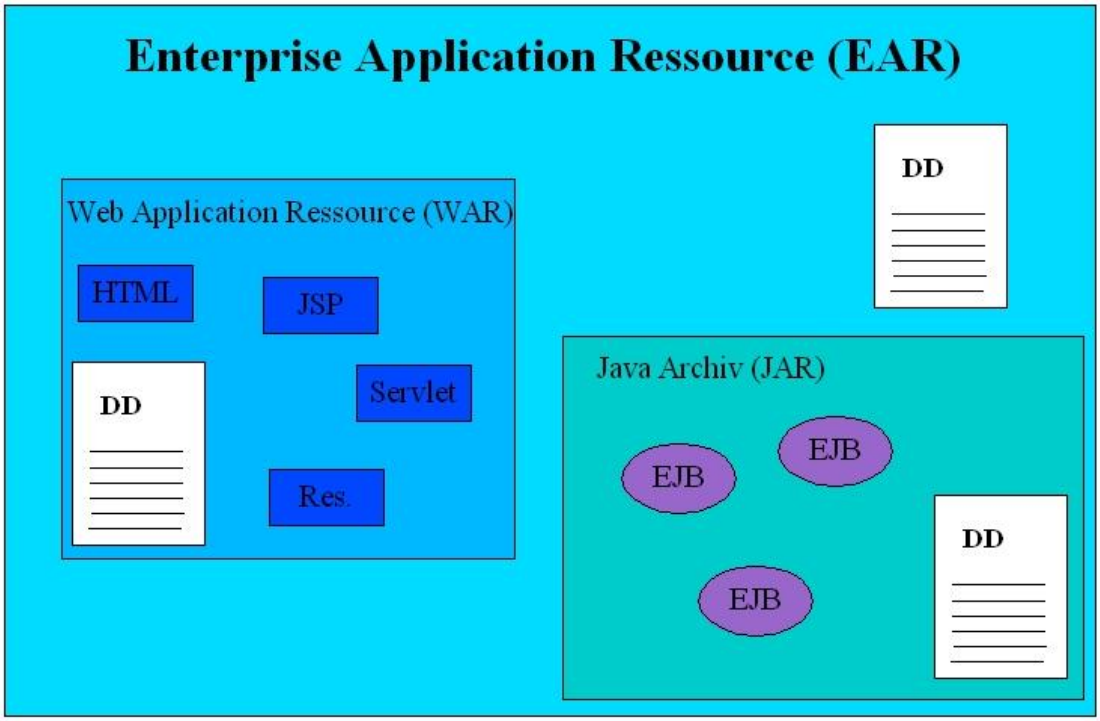


Abbildung 4.1 : Komponenten einer J2EE – Anwendung

Auch hier soll im folgenden Abschnitt noch kurz auf den Deployment Deskriptor eines EAR – Archivs eingegangen werden.

4.3.1 application.xml

Im Falle der application.xml – Datei gibt es nicht viel zu sagen. Sie ist wie ihre beiden Vorgänger eine XML – Datei, welche nur die Namen der Enterprise – Anwendung und deren Bestandteile enthält.

- *application* der Anfang des Deployment Deskriptors, enthält alle EAR – relevanten Informationen
- *display-name* Name der Anwendung
- *module* enthält Angaben zu einem bestimmten Archiv, dieses wird dann in das EAR eingebunden
  - ➔ *web* gibt die Art des Moduls an, entweder WAR oder JAR, hier WAR
    - web-uri Angabe des Archivnamens
    - context-root unter diesem Namen ist die Anwendung später zu erreichen
  - ➔ ejb gibt die Art des Moduls an, entweder WAR oder JAR, hier JAR

```
<?xml version="1.0" ?>
<!DOCTYPE application (View Source for full doctype...)>
<application>
 <display-name>WOMBank</display-name>
 <module>
  <web>
   <web-uri>WOMBankOS390WAR.war</web-uri>
   <context-root>/WOMBank</context-root>
  </web>
 </module>
 <module>
  <ejb>WOMBankOS390EJB.jar</ejb>
 </module>
  ...
</application>
```

5 Erste Schritte

Neben der Entwicklung und Implementation der WOMBank – Anwendung mit dem J2EE – basierten Ansatz, sollten auch mehrere Tutorien erarbeitet werden, welche auf einfache und verständliche Weise den Umgang mit den Session und Entity Beans im Zusammenhang mit dem WebSphere Application Server erläutern. Als Grundlage für diese begleitenden Übungen hat, wie in so vielen anderen Fällen, das Hello World Programm eine zentrale Rolle gespielt. Dazu wurde am Anfang mit Hilfe eines Java – Servlets der Text „Hello World!“ ausgegeben werden. Später wurde darauf aufbauend eine Session Bean und eine Entity Bean in die Applikation integriert. Da die gesamten Anwendungen auf einem Entwicklungssystem, welches sich in deutlicher Weise von der Einsatzumgebung unterscheidet, erstellt wurden, werden im nächsten Abschnitt die beiden Systeme beschrieben und auf deren Eigenschaften kurz eingegangen werden.

5.1 Beschreibung der Entwicklungs – und der Einsatzumgebung

Das Entwicklungssystem war mein eigener Rechner mit dem Microsoft Betriebssystem Windows XP, dem IBM WebSphere Application Server (WAS) for Windows in der Version 5.1 und eine IBM DB2 Universal Database in der Version 8. Diese beiden Produkte konnte man sich vollständig und voll funktionsfähig von der IBM Webseite herunterladen, in dem man als Verwendungszweck „studentische Arbeit“ angegeben hat. Des Weiteren wurde die Java Standard Edition (J2SE) in der Version 1.4.2 und die Java Enterprise Edition in der Version 1.4 installiert.
Mit dieser Konfiguration und dem komfortablen Web – Frontend des IBM WAS 5.1 war es sehr schnell möglich die implementierte WOMBank – Anwendung zu installieren und zum Laufen zu bringen.

Das Einsatzsystem war ein Rechner mit dem z/OS – Betriebssystem von IBM in der Version 1.4. Dieses System bringt folgende Voraussetzungen mit sich : IBM WebSphere Application Server Version 4.0 , welcher als PlugIn im HttpServer 5.3 vorlag. Die IBM DB2 Database in der Version 7 übernimmt die Datenhaltung auf diesem System. Damit wurden die Standards J2SE in der Version 1.3 und J2EE in der Version 1.2 und damit die EJB – Spezifikation 1.1 unterstützt.
Die Entwicklung auf einem Windowssystem war notwendig, da der IBM WAS auf dem z/OS – Rechner nur mit Hilfe des sogenannten System Management User Interfaces, kurz SMUI, angesprochen werden kann. Diesen Windows – Client kann man sich per FTP von dem Rechner (in diesem Fall Yoda) herunter laden, auf dem sich der WAS befindet. Nach der Installation und der Einrichtung stehen einem alle Funktionen des Application Servers zur Verfügung. Darauf wird in einem späteren Kapitel eingegangen.
Für die Installation einer Anwendung wird noch ein weiteres Tool benötigt, welches die Anwendung

deployed. Das bedeutet, es werden die noch benötigten Deployment Deskriptoren erstellt und dem EAR – Archiv eventuell benötigte Class – Dateien hinzugefügt. Dieses Werkzeug wird ebenfalls auf der IBM Webseite zum Download angeboten.
Nach einigen Problemen aufgrund der ungewohnten und unbekannten Arbeits – und Vorgehensweise war es auch auf dem z/OS – System möglich die WOMBank – Anwendung nach einigen Änderungen zu installieren und zum Funktionieren zu bringen.

5.2 Hello World im Internet

Um sich die grundlegenden Funktionen und den Umgang mit dem IBM WAS verständlich zu machen, wurde zunächst eine einfache „Hello World!“ – Applikation entwickelt. Nachdem diese Anwendung, welche nur aus einem Servlet bestand, erfolgreich seinen Dienst verrichtete, wurde eine Session Bean und später auch eine Entity Bean hinzugefügt.

5.2.1 HelloWorldServlet

Diese simple Anwendung besteht nur aus einem Servlet, welches ganz einfach den Text „Hello World!“ im Browser – Fenster ausgibt, wenn man es aufruft. Wie aus dem Quelltext weiter unten zu erkennen ist, wird einfach eine HTML – Seite erzeugt, um den gewünschten Text anzuzeigen. Um die Übersichtlichkeit zu bewahren, wird nur in diesem Beispiel jede Zeile extra von dem PrintWriter ausgegeben. Normalerweise würde der gesamte Text zu einer Zeile zusammengefasst und dann ausgedruckt.

```
package ycubas.servlet;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorldServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
         response.setContentType("text/html");
         PrintWriter out = response.getWriter();
         try{
                  out.println("<html><head>");
                  out.println("<title>Hello World!</title>");
                  out.println("</head>");
                  out.println("<body>");
                  out.println("<h1>Hello World!</h1>");
```

```
                out.println("</body></html>");
            }catch(Exception e){out.println(e.toString());}
    }
}
```

Wie in der nächsten Abbildung ersichtlich ist, besteht diese Version der „Hello World“ – Anwendung
nur aus einer WAR – Datei, welche, neben dem Deployment Deskriptor, das vorher beschriebene Servlet beinhaltet.

Abbildung 5.1 : Aufbau der Hello World Anwendung Version 1

In der web.xml wird die URL spezifiziert, mit der nach der Installation das Servlet aufgerufen werden kann.

```
<servlet-mapping>
   <servlet-name>HelloWorldServlet</servlet-name>
   <url-pattern>/HelloWorldServlet</url-pattern>
</servlet-mapping>
```

Weiterhin wird der Pfad des Servlets durch den Context Root der Anwendung bestimmt. Das ist der Teil der URL, der dem Application Server mitteilt, welche Anwendung überhaupt gemeint ist, da es sehr wahrscheinlich ist, dass gleichzeitig mehrere Applikationen auf dem WAS installiert sind. Diese eindeutige Bezeichnung steht in der application.xml, dem Deployment Deskriptor des EAR – Archives.

```
<display-name>Hello World</display-name>
 <module>
   <web>
     <web-uri>HelloWorldSession.war</web-uri>
     <context-root>/Test</context-root>
   </web>
 </module>
```

Danach wird das Servlet später unter folgender URL zu erreichen sein:
http://techspru2.informatik.uni-leipzig.de/Test/HelloWorldServlet

5.2.2 Hello World mit Session Bean

Der nächste logische Schritt, um die Komplexität der Anwendung ein wenig zu erweitern, ist die Integration einer Session Bean. Die Funktionsweise einer solchen Bean wird in dem Abschnitt 3.3 und den folgenden Unterabschnitten genau erläutert. Sie wird über zwei verschiedene Schnittstellen angesprochen, dem sogenannten Home – Interface, wodurch alle den Lebenszyklus einer Bean betreffenden Aufgaben angesprochen werden können, und dem Client – Interface, mit dessen Hilfe die Funktionen der Beanklasse aufgerufen werden.

Die HelloWorld Session Bean hat nur eine einfache Funktion und ist nicht besonders komplex. Sie bekommt als Parameter einen String von einem Servlet übergeben und fügt diesem nur einen weiteren String hinzu, bevor dieser wieder zurückgeschickt wird. Das könnte ungefähr so aussehen: Nachdem ein Text eingegeben wurde, wird dieser, der zum Beispiel „Hello World“ lauten könnte, an die Session Bean weitergeleitet. Diese fügt dem String noch die Zeichenfolge „you said: “ hinzu und gibt diesen zurück an das Servlet, welches dann folgendes ausgibt:

you said: Hello World

Damit dies alles funktioniert, müssen Veränderungen an dem Servlet des Hello – World – Programms vorgenommen werden. Um die Funktionalität der ersten Version beizubehalten, wird ein komplett neues Servlet auf der Grundlage des Ersten implementiert.
Es muss ein Eingabefeld zur Verfügung gestellt werden, welches die Möglichkeit bietet einen Text einzugeben. Des Weiteren müssen die nötigen Programmroutinen integriert werden, um die Session Bean aufrufen und deren Funktionen nutzen zu können. Da das Servlet dadurch wesentlich komplexer wird, wird in dieser und in zukünftigen Versionen sehr großer Wert auf Übersichtlichkeit und Wiederverwendbarkeit gelegt. Die folgenden Programmzeilen ermöglichen die Umsetzung der eben erwähnten Veränderungen.

```
protected void printHome(HttpServletResponse res) {
      String sHtml =
```

```
        "<br>Geben Sie eine Nachricht ein, welche dann an eine Session Bean geleitet
wird.<br>";
      sHtml += "<form name='Message Input' method='get' action='HelloWorldServlet2'>";
      sHtml += "<input name='message' size=20 maxlength=60 value=''><br>";
      sHtml += "<input type='submit' value='Submit'><input type='reset'
value='Reset'></form>";
      print(res, sHtml);
   }


try{
        InitialContext ctx = new InitialContext();
        HelloWorldHome hwHome =
                (HelloWorldHome)PortableRemoteObject.narrow
                        (ctx.lookup"Diplomarbeit/HelloWorldSession"),HelloWorldHome.class);
        HelloWorld hw =
                (HelloWorld)PortableRemoteObject.narrow(hwHome.create(),
HelloWorld.class);
        print(res,"HelloWorld Stateless Session Bean ..." + hw.hello(message));
        }catch(NamingException e){
                print(res,e.toString());
        }catch(CreateException e){
                print(res,e.toString());
        }
```

Es wird zuerst im InitialContext nach dem JNDI – Namen der Bean gesucht und danach das Home – Interface angesprochen, um eine Instanz der Bean zu erzeugen. Diese Mechanismen werden, wie vorhin schon erwähnt, in dem Abschnitt 3.3 und dessen Unterabschnitten genau erläutert. Danach wird nur noch die Methode *hello()* mit dem dazugehörigen Parameter *message*, also der Zeichenkette, aufgerufen und der bearbeitete Text ausgegeben.

In der folgenden Abbildung wird noch einmal kurz die Struktur der neuen Applikation aufgezeigt. Das Web – Archiv bleibt soweit erhalten. Es wird nur noch ein weiteres Servlet und die dazugehörige Konfiguration hinzugefügt. Ausserdem wird noch das Java Archiv integriert, welches alle benötigten Klassen für die Session Bean enthält. Das sind das Home – Interface HelloWorldHome.class, das Client – Interface HelloWorld.class und die eigentlich Beanklasse HelloWorldBean.class.

Abbildung 5.2 : Aufbau der Hello World Anwendung Version 2 mit Session Bean

Nach den Angaben in den Deployment Deskriptoren wird das zweite Servlet später unter folgender URL zu erreichen sein:

http://techspru2.informatik.uni-leipzig.de/Test/HelloWorldServlet2

5.2.3 Hello World mit Entity Bean

Nachdem die Hello World Anwendung mit der Session Bean fehlerfrei arbeitet, ist der nächste Schritt, eine Entity Bean mit deren Möglichkeiten zur Speicherung persistenter Daten in die Applikation einzubinden. Der dazu nötige Aufwand ist allerdings um einiges größer als der, welcher für eine Session Bean nötig ist. Außerdem muss dazu noch angemerkt werden, das in verschiedenen Bereichen auf dem Einsatzsystem Administratorrechte benötigt werden, um Einstellungen vornehmen und eine Tabelle in der Datenbank erstellen zu können. Deshalb ist es wahrscheinlich, das ein solches Tutorium nicht für einen Übungsbetrieb geeignet ist.

Mit Hilfe der Entity Bean ist es möglich, Daten in einer Datenbank vorzuhalten, um diese zu einem späteren Zeitpunkt, wenn sie wieder benötigt werden, zur Verfügung zu stellen. In dem Hello World Szenario könnte eine solche Funktionsweise wie folgt genutzt werden:
An die Bean wird wie bei der vorherigen Variante ein Text übergeben. Nun gibt es zwei Möglichkeiten. Entweder es befindet sich schon ein Eintrag in der Datenbank oder eben noch nicht. Sollte der erste Fall zutreffen, wird der schon vorhandene Datensatz ausgelesen und der als Parameter übergebene String wird an dessen Stelle in die Datenbank eingetragen. Nun kann das Servlet den eben eingegebenen Text und den gespeicherten

Text gemeinsam ausgeben. Im zweiten Fall wird der eingegebene Text in die Datenbank gespeichert und nur dieser wird ausgegeben. Beim nächsten Aufruf wird dann der erste Fall eintreten. Diese Vorgehensweise könnte man im einfachsten Sinne mit einer History vergleichen. Eine Ausgabe des Servlets könnte dann wie folgt lauten:

HelloWorld Entity Bean ... this time you said Hello World, last time you said Hello Europe.

Auch in diesem Fall wird ein neues Servlet auf der Grundlage des Vorgängers programmiert. Allerdings sind die Änderungen minimal. Lediglich der Aufruf der Bean und der Methoden unterscheiden sich ein wenig von denen der Session Bean. Die Entity Bean hat für das Schreiben in und das Auslesen aus der Datenbank jeweils eine Methode pro Variable. In diesem Fall wäre das eine setGreeting() – Methode, um den Eintrag persistent zu speichern und eine getGreeting() - Methode, um die Nachricht wieder auszulesen.

```
try{
        ...

        hwe = (HelloWorldEntity)PortableRemoteObject.narrow(hweHome.findByPrimaryKey
((new          HelloWorldEntityPK("0"))),HelloWorldEntity.class);
        print(res,"HelloWorld Entity Bean ... this time you said " + message + ",last time you
said          "+hwe.getGreeting());
        hwe.setGreeting(message);
        ...
        }catch(FinderException e){
                try{
                        print(res,"Create...<br>");
                        hwe = (HelloWorldEntity)PortableRemoteObject.narrow
(hweHome.create                         (message),HelloWorldEntity.class);
                        print(res,"HelloWorld Entity Bean ... you said "+hwe.getGreeting());
                        }catch


                        ....
```

Wie in dem Quelltext zu erkennen ist, wird die Unterscheidung, ob sich schon ein Datensatz in der Datenbank befindet, mittels einer FinderException getroffen. Diese wird geworfen, wenn der entsprechende Datensatz nicht in der Datenbank vorhanden ist.

In der folgenden Abbildung wird die endgültige Struktur der Hello – World – Anwendung aufgezeigt. Es wurde ein weiteres Servlet und ein Java Archiv, welches alle Beanklassen der

Hello World Entity Bean enthält, hinzugefügt.

Abbildung 5.3 : Aufbau der Hello World Anwendung Version 3 mit Entity Bean

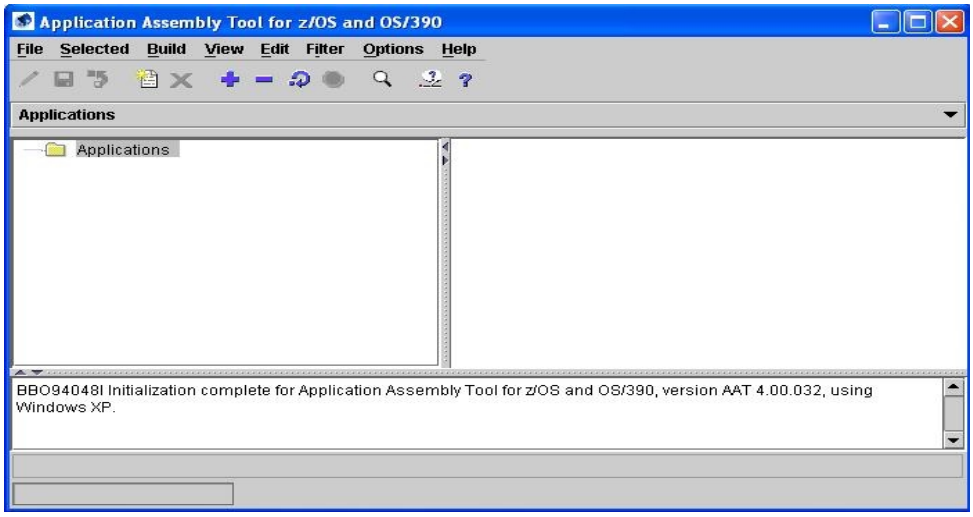
Das Servlet, welches die Entity Bean aufruft, wird nach der Installation auf dem WebSphere Application Server unter folgender URL zu erreichen sein:

http://techspru2.informatik.uni-leipzig.de/Test/HelloWorldServlet3

5.2.4 Installation der Anwendung

Bevor die Applikation auf dem WAS installiert werden kann, muss diese noch deployed werden. Das bedeutet, dass noch eventuell fehlende Deployment Deskriptoren erstellt werden und Klassen, die für die CORBA – Anbindung wichtig sind, hinzugefügt werden. Dafür wird das Assembly Tool von IBM verwendet, welches sich im Verzeichnis **Diplom Thomas Kumke\Software\AssemblyTool** auf der CD befindet, die dieser Diplomarbeit beliegt. Nach der Installation und dem Start der Anwendung wird das folgende Fenster geöffnet. Dies kann einige Sekunden dauern.

Abbildung 5.4 : Startbildschirm des Assembly Tools



Um die EAR – Datei zu importieren, wird mit der rechten Maustaste auf den mit grau unterlegten Begriff *Applications* geklickt. Im darauffolgenden Menü wird der Menüpunkt *Add* gewählt. Danach kann man die Anwendung im Dateimenü auswählen. Nach dem Import stehen verschiedene Funktionen zur Verfügung. Zum Deployen wird dieses Mal mit der rechten Maustaste auf den Applikationsnamen geklickt. Im Menü ist nun der Punkt *Deploy* erschienen, welcher das Deployen einleitet. Dieser Vorgang kann bei größeren Anwendungen einige Minuten dauern. Zum Abschluss muss das Archiv nur noch exportiert werden. Der Menüpunkt sollte nach dem Deployen zur Verfügung stehen.
Die Applikation ist nun für die Installation auf dem Application Server bereit. Dazu wird das Administration – Tool, der sogenannte SMUI – Client, aufgerufen. Kurz nach dem Start erscheint ein Loginformular in dem die folgenden Parameter eingetragen werden. Das Passwort ist CBADMIN.

Abbildung 5.5 : Loginparamater für den SMUI - Client

Bevor der Startbildschirm des Administrationstools erscheint, kann einige Zeit vergehen. Das Laden aller vorhandenen Konversationen, so werden die Sessions genannt , in denen Veränderungen am Zustand des WebSphere Application Servers vorgenommen werden, dauert dabei besonders lange. Um eine J2EE – Applikation installieren zu können, muss eine neue Konversation angelegt werden. Dazu dient der Menüpunkt *Add Conversation*, nach dessen Anklicken man zur Eingabe eines Namens gebeten wird. Im Unterpunkt *J2EEServers* besteht nun die Möglichkeit eine J2EE – Applikation zu installieren.

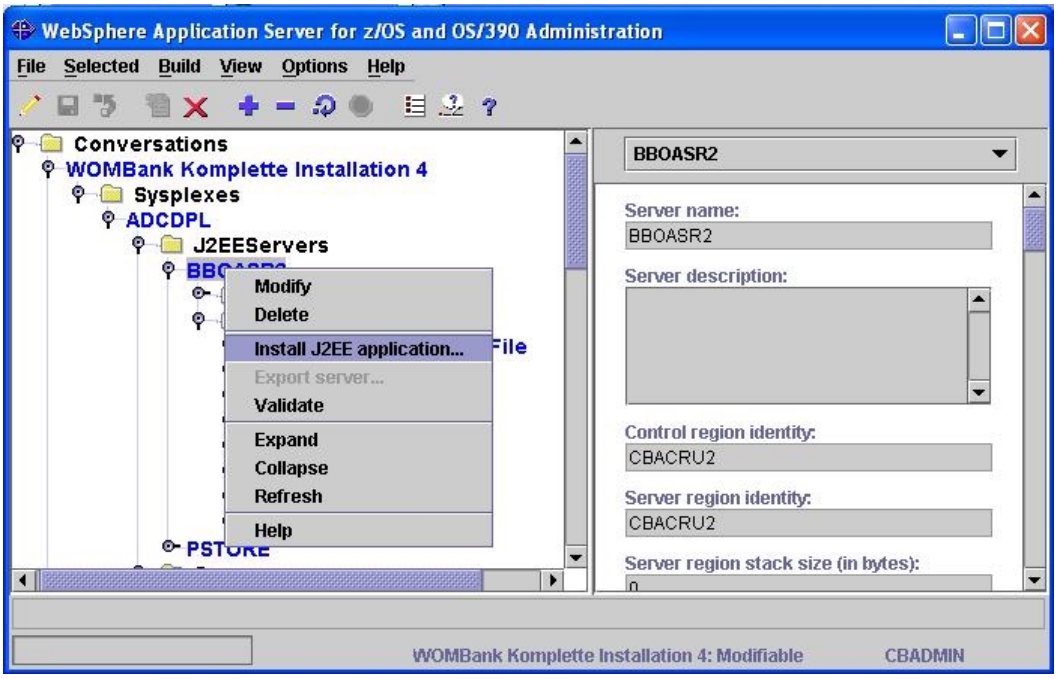


Abbildung 5.6 : SMUI- Installation einer J2EE - Applikation

Nachdem eine Anwendung ausgewählt wurde, wird diese via FTP auf das Einsatzsystem übertragen und dort installiert. Vorher könnte es aber sein, das noch einige Einstellungen vorgenommen werden müssen. So muss zum Beispiel den Entity Beans noch eine Datenbankverbindung oder einigen Beans noch ein JNDI – Namen zugewiesen werden. Dies geschieht in einem Bildschirm, der in Abbildung 5.7 dargestellt ist. Dabei sind in allen noch nicht mit einem grünen Haken gekennzeichneten Feldern Einstellungen vorzunehmen. Sind diese vorgenommen, besteht die Möglichkeit, nochmals das EAR zu speichern oder mit der Installation fortzufahren. Zum Abschluss muss die Konversation nun noch bestätigt (*engl.* commit) und anschliessend aktiviert werden. Nun steht die Anwendung zur Verfügung. Weitere Hinweise und eine genaue Schritt – für – Schritt – Anleitung zur Installation sind in den Niederschriften der beiden Tutorien und in den Dokumenten [7],[8] und [9] enthalten,

welche dem WebSphere Application Server und dieser Arbeit beiliegen.

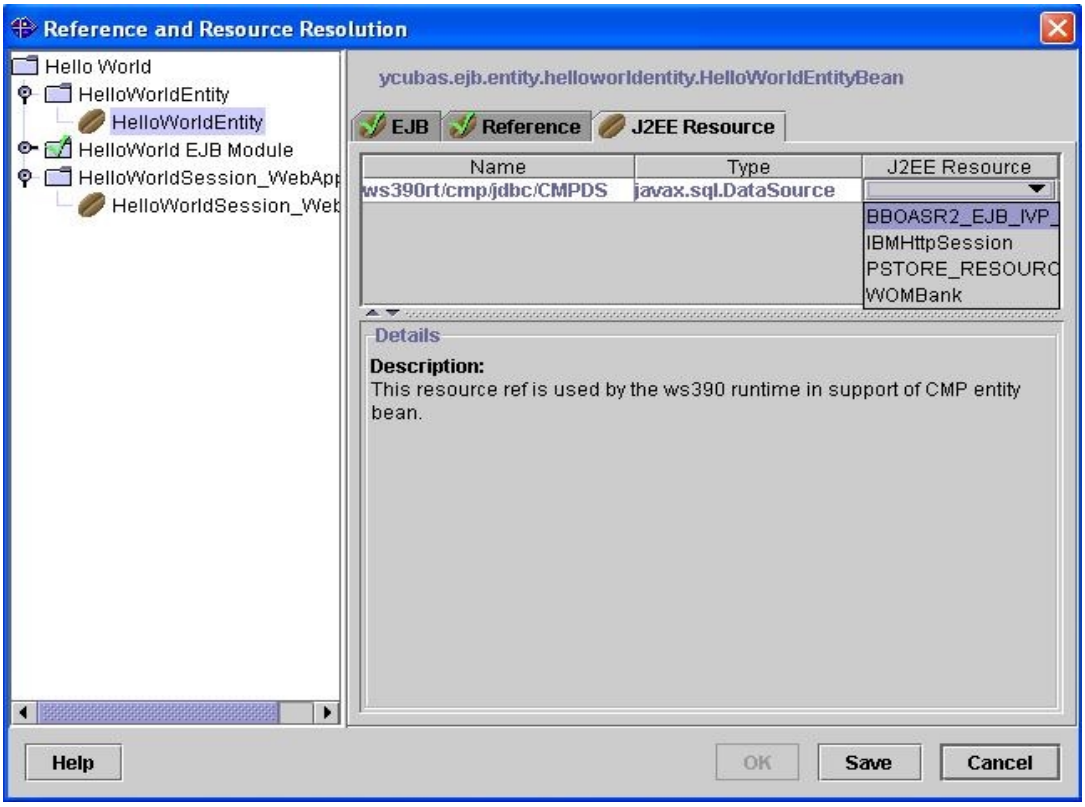


Abbildung 5.7 : Resource Resolution im SMUI

Nachdem die Installation abgeschlossen ist, sind noch weitere Aufgaben zu erledigen, bevor die Anwendung einwand – und fehlerfrei funktioniert. Auf dem Einsatzsystem sind nun noch die Tabellen, die von den Entity Beans angesprochen werden, in der DB2 – Datenbank zu erstellen. Dabei werden allerdings Administratorrechte benötigt, da die Tabellen unter einem anderem Nutzer erstellt werden müssen. Das läßt sich folgendermaßen erklären. Der J2EE – Server, welcher für die Installation von J2EE – Anwendungen vom WAS bereitgestellt wird, läuft unter einer bestimmten Nutzer – Gruppe, in der sich der User CBASRU2 befindet. Unter diesem Account laufen alle Tabellen, die in der DB2 für den Betrieb des Application Servers benötigt werden. Installiert man nun eine neue Entity Bean, muss man unter diesem Nutzer eine neue Tabelle mit den Bean – Daten erstellen. Diese Rechte hat man mit einem gewöhnlichen Account nicht.

Es ist auch möglich, einen neuen J2EE – Server zu erstellen. Dies wäre allerdings wesentlich aufwendiger, da für jeden Account ein neuer Server erstellt werden müßte, auf

den der User dann Zugriff hat. Das wäre der Performance des WAS allerdings nicht besonders dienlich.
Gesetz dem Fall, dass diese Rechte dem Account zur Verfügung stehen und eine Entity Bean in eine Anwendung integriert wurde, wie bei der Hello – World – Anwendung Version 3, kann man nun noch die dazugehörige Tabelle in der Datenbank erstellen.

Um mit Hilfe des eigenen Rechners auf den z/OS – Server zugreifen zu können, benötigt man einen 3270 – Klienten. Dieser kommuniziert über das tn3270 Übertragungsprotokoll (siehe auch [2]) mit dem z/OS – Communication Server. Somit hat man Zugriff auf alle wichtigen Dienste, so auch auf das DB2 Administrationswerkzeug. Unter Windows ist der QWS3270 – Emulator, welcher im folgenden auch verwendet wird, ein zuverlässiges und einfaches Tool. Dieser liegt neben einer Installationsanleitung auf dem Jedi – Rechner des Instituts für Informatik zum Download bereit. Nach der Installation wird ein Startbildschirm angezeigt. Um sich mit dem z/OS – Rechner zu verbinden, muss der Menüpunkt *Connect* gewählt werden. Danach wird folgendes Fenster sichtbar:

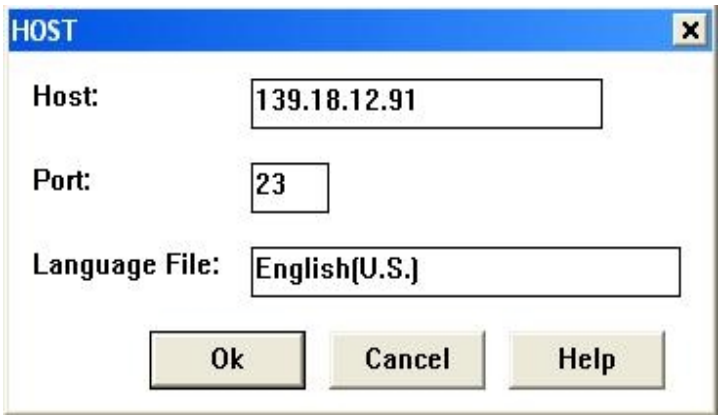


Abbildung 5.8 : Loginparameter des QWS3270 – Clients

Je nach der Konfiguration des QWS3270 – Emulators müssen die Parameter wie in der Abbildung eingetragen beziehungsweise so verändert werden. Nach dem Bestätigen durch den OK – Button wird eine Verbindung zu dem z/OS – Rechner aufgebaut und es ist ein Begrüßungsbildschirm zu sehen. Um sich in das TSO – Subsystem einzuloggen, welches die benötigten Dienste und Werkzeuge zur Verfügung stellt, wird in diesem Fenster *L TSO* eingegeben. Nach erfolgreicher Eingabe des Usernamens und erfolgter Bestätigung durch das Password, wird eine Übersicht aller möglichen Optionen angezeigt, welche abhängig

von den an den Account gebundenen Rechten sind. Durch die Eingabe der Option M, wird das erweiterte Menu wie in folgender Abbildung dargestellt.

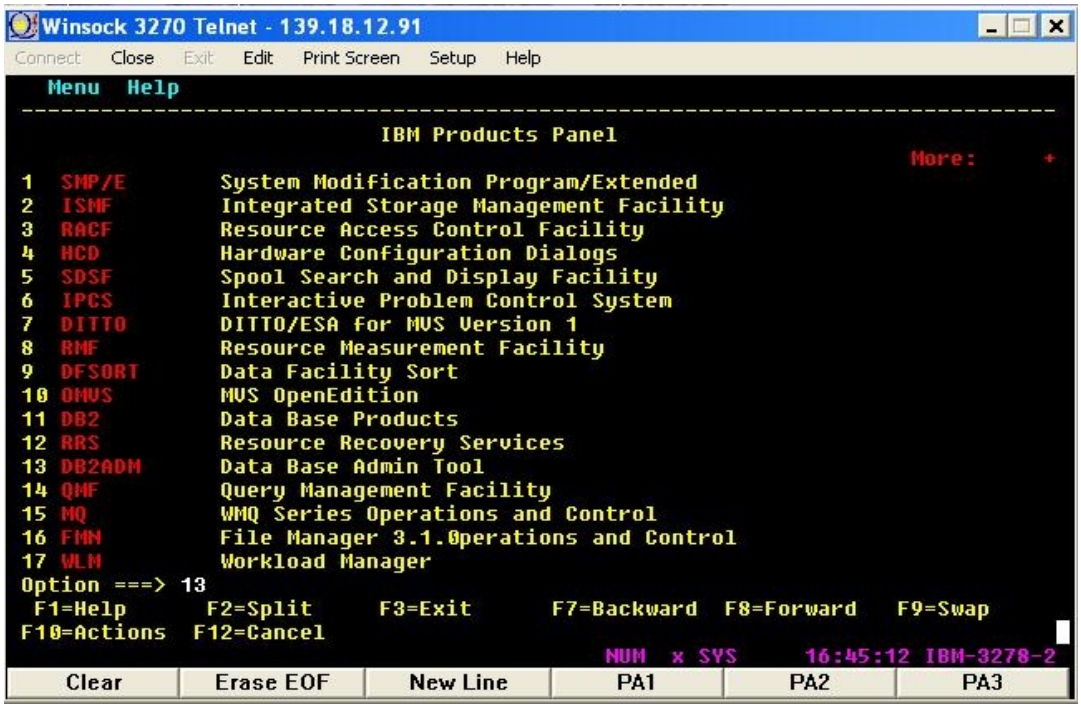


Abbildung 5.9 : erweitertes Menu im TSO - Subsystem

Um die Tabelle in der Datenbank zu erstellen, wird, wie in der oberen Darstellung zu erkennen, das Admin Data Base Tool aufgerufen. Mit diesem hat man die Möglichkeit alle die Datenbank betreffenden Einstellungen und Konfigurationen vorzunehmen, die erforderlichen Rechte vorausgesetzt. Im Administrationsbildschirm ist die Option 2 (Execute SQL Statement) und danach die Option 1 (Execute SQL Statement from Screen Input) auszuwählen. Damit gelangt man in den Eingabebereich, in dem folgende Zeilen eingetragen werden. Durch das Betätigen der Enter – Taste wird die Ausführung der SQL – Befehle eingeleitet.

```
CREATE TABLE CBASRU2.HELLOWORLDENTITY
    (OID VARCHAR(250) NOT NULL,
     GREETING VARCHAR(250),
     CONSTRAINT HELLOWORLDENTITYPK PRIMARY KEY(OID))
        IN ESTOREDB.ESTORETS;COMMIT;

CREATE UNIQUE INDEX HELLOWORLDENTITIX
```

```
ON CBASRU2.HELLOWORLDENTITY(OID ASC);COMMIT;
```

Mit diesen SQL Statements wird eine Tabelle Namens HELLOWORLDENTITY für den Benutzer CBASRU2 erstellt. Sollte diese Tabelle für den Nutzer schon vorhanden sein, führt das Ausführen der Befehle natürlich zu einer Fehlermeldung.

Die Anwendung sollte nun fehlerfrei funktionieren.

5.2.5 Test der HelloWorld - Anwendung

Beim Aufruf der HelloWorld – Anwendung mit folgender URL wird der in Abbildung 5.10 dargestellte Anfangsbildschirm angezeigt. Durch Anklicken der Links gelangt man zu den einzelnen Teilen der Applikation.

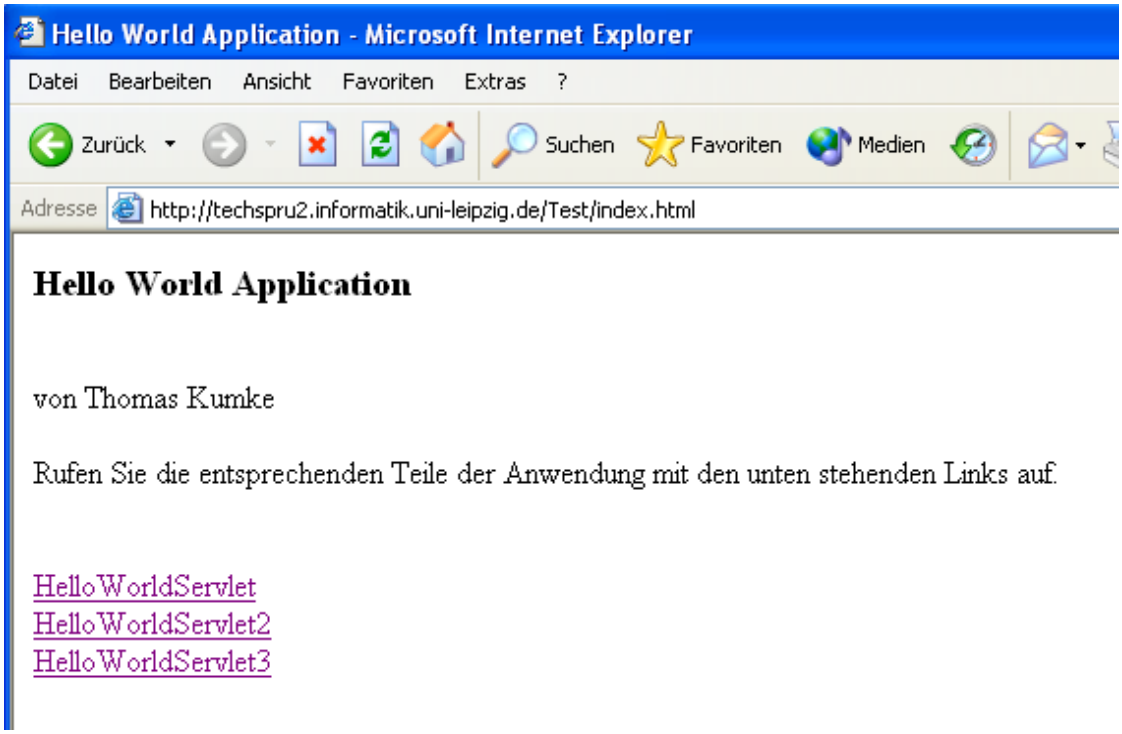


Abbildung 5.10 : Startbildschirm der HelloWorld – Anwendung

6 Die WOMBank – Anwendung

Das Ziel dieser Diplomarbeit war es die schon vorhandene WOMBank – Anwendung aus der Diplomarbeit von Herrn Ralf Ronneburger „Internet – basierte Anwendungen mit Java und DB2 unter OS/390“ so weit zu verändern, dass Sie komplett mit Enterprise Java Beans arbeitet. Das bedeutete Umbau und Neuentwicklung der Applikation von der Basis an. Nach außen hin bleiben die Veränderungen verborgen, da die Präsentationsschicht dabei größtenteils erhalten blieb. Nur die Geschäftslogik und die Datenhaltung wurden neu erarbeitet und anschliessend implementiert. Im Laufe der Umstrukturierung der Anwendung wurden fünf Entity Beans zur persistenten Datenhaltung, drei Session Beans zur Umsetzung der Geschäftslogik, mehrere Servlets zum Testen, sowie das Servlet zur Steuerung der Anwendung entwickelt. Auch der Vorgang des Einloggens wurde, vom programmiertechnischen Aufwand gesehen, stark vereinfacht. Ein sessionbasierter Ansatz, in dem sitzungsrelevante Daten in der Session selbst aufbewahrt werden, dient nun als Login - Mechanismus. Weiterhin wurde eine umfangreiche JavaDoc – Dokumentation erstellt, welche die einzelnen Klassen und Methoden auf einen Blick beschreibt und während der Benutzung der Anwendung aufgerufen werden kann. Weitere Informationen zu den Veränderungen werden in den folgenden Abschnitten geliefert.

6.1 Struktur und Aufbau der WOMBank – Anwendungen

Im Folgenden wird auf den Aufbau der beiden WOMBank – Anwendungen eingegangen. Eine einfache Unterscheidung der Struktur erfolgt nach den in Kapitel 2.3.1 angegebenen n – tier – Architektur – Ansatz. Dazu wird zuerst der servletbasierte Ansatz genauer betrachtet und im Anschluss auf die Umsetzung mit Hilfe der Enterprise Java Beans eingegangen.

6.1.1 servletbasierte WOMBank – Applikation

Die Struktur dieser Anwendung ist sehr einfach gestrickt. Sie besteht nur aus einem einzigen

Servlet, welches sowohl die Geschäftslogik als auch die Koordination zur Umsetzung der persistenten Datenhaltung beinhaltet. Weiterhin gehören die für Darstellung in einem Webbrowser benötigten HTML – und Bilddateien mit zu dem Umfang der Applikation. Da die gesamten Anwendungsregeln im Servlet enthalten sind und die Datenbankzugriffe ebenfalls vom Servlet selbst organisiert werden, ist diese Datei dementsprechend groß. Eine solche Umsetzung ist heutzutage nicht mehr üblich, da spätere Änderungen durch die schlechte Modularisierung und Skalierbarkeit mit einem hohen Aufwand verbunden sind. Allerdings ist diese Anwendung durch die geringe Komplexität und der einfachen Struktur noch recht überschaubar. Bei größeren Projekten würde die Übersichtlichkeit unter diesem Ansatz leiden. Für weitergehende Informationen zu der WOMbank – Anwendung wird angeraten in der Diplomarbeit „Internet – basierte Anwendungen mit Java und DB2 unter OS/390“ von Herrn Ralf Ronneburger nachzulesen.

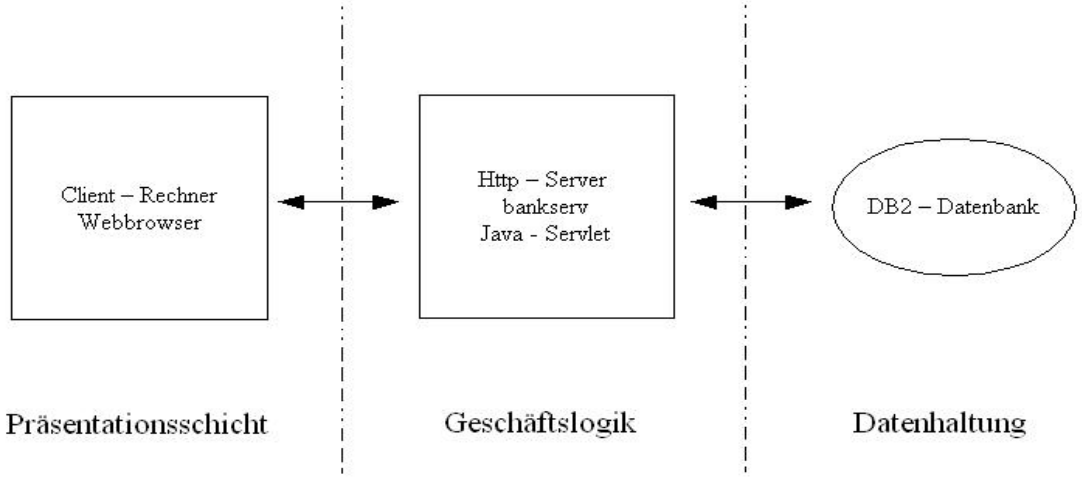


Abbildung 6.1 : 3 – tier – Architektur der servletbasierten WOMBank – Anwendung

Nach dem n – tier – Ansatz bezeichnet man diese Applikation ganz klar als eine 3 – tier – Architektur. Die Präsentationsschicht wird auf dem Clientrechner über den Webbrowser realisiert, der Httpserver auf dem z/OS – Rechner übernimmt die Umsetzung der Geschäftslogik und die dritte Schicht, also die Datenhaltung an sich, wird von der DB2 – Datenbank übernommen. In Abbildung 6.1 wird dieser Zusammenhang übersichtlich dargestellt.

Wie aus der Abbildung 6.2 zu entnehmen ist, liegt die Präsentationsschicht in statischen HTML – Dokumenten vor. Da aber Kontostände und Benutzerdaten dynamisch in diese

Seiten integriert werden müssen, bedient man sich eines Tricks. Innerhalb der statischen Dokumente befinden sich Platzhalter, welche vor der Darstellung im Webbrowser durch die entsprechenden Daten aus der Datenbank ersetzt werden. Dazu wird eine eigens von IBM entwickelte Methode verwendet, die allerdings in dem EJB – basierten Ansatz der WOMbank keine Verwendung mehr findet, da man dafür noch weitere Java – Packages in die Anwendung integrieren müßte.

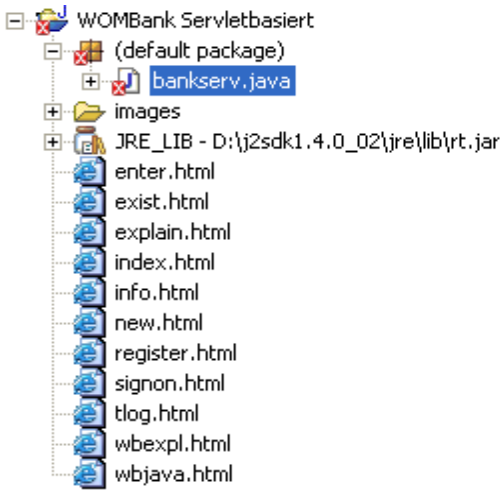


Abbildung 6.2 : Komponenten des servletbasierten WOMBank – Ansatzes

6.1.2 EJB – basierter Ansatz der WOMBank

Bei der Umsetzung der WOMBank – Applikation wurden komplexere Strategien verfolgt. Für die Datenhaltung ist immer noch eine DB2 – Datenbank verantwortlich, die Organisation der persistenten Datenspeicherung übernehmen nun aber Enterprise Java Entity Beans. Für das reibungslose Funktionieren der Anwendung sind fünf Tabellen in der Datenbank nötig, demnach also auch die selbe Anzahl an Entity Beans. Das sind die WBPassWd – Entity Bean, die für die Speicherung des Usernamens und des Passwortes verantwortlich ist, die WBBankAccount – Entity Bean, welche die persistente Datenhaltung aller das Bankkonto betreffenden Daten übernimmt, die WBALogRecord – Entity Bean, deren Funktion es ist, alle Aktionen durch entsprechende Loggingdaten festzuhalten und zwei weiteren Entity Beans. Das ist zum einen die WBNumber – Entity Bean, welche für eine fortlaufende Nummerierung der Bankkonten und der Logging – Datensätze verantwortlich ist. Die Zweite ist die ByteContainer – Entity Bean, die , wie der Name schon sagt, für die Aufnahme von

Bytedaten bestimmt ist. Dazu gehören unter anderem die gesamten HTML – Seiten, die nicht mehr in der Anwendung selber integriert sind.
Die gesamte Abwicklung der Anwendungsregeln wurde ebenfalls ausgelagert und in Session Beans untergebracht. Man kann die Geschäftsprozesse grob in drei Typen einteilen und somit sind drei verschiedene Session Beans entstanden. Das ist zum einen die HandleLogin – Session Bean, die mit Ihren Methoden dafür sorgt, das das Login und Logout einwandfrei funktioniert. Die HandleAccount – Session Bean dient der Handhabung aller das Bankkonto betreffenden Funktionen und zu guter Letzt die HandleLogging Session Bean, welche für jede Aktion des Nutzers einen Logging – Eintrag erzeugt. Ganz einfach gesagt, jede Entity Bean wird von einer bestimmten Session Bean angesprochen.
Das WOMBankServlet an sich dient nur noch als Schnittstelle zwischen der Präsentationsschicht und der Applikationslogik, indem es je nach Parameterübergabe entscheidet, welche Session Bean aufzurufen ist. Im weiteren Verlauf der Implementierung wurden noch zusätzliche Test – Servlets entwickelt, um während dieser Phase den Fortschritt der Umsetzung zu begutachten und eventuelle Fehler besser ausmachen zu können. Alle zu der WOMBank – Applikation gehörenden Komponenten sind in der Abbildung 6.3 aufgelistet.

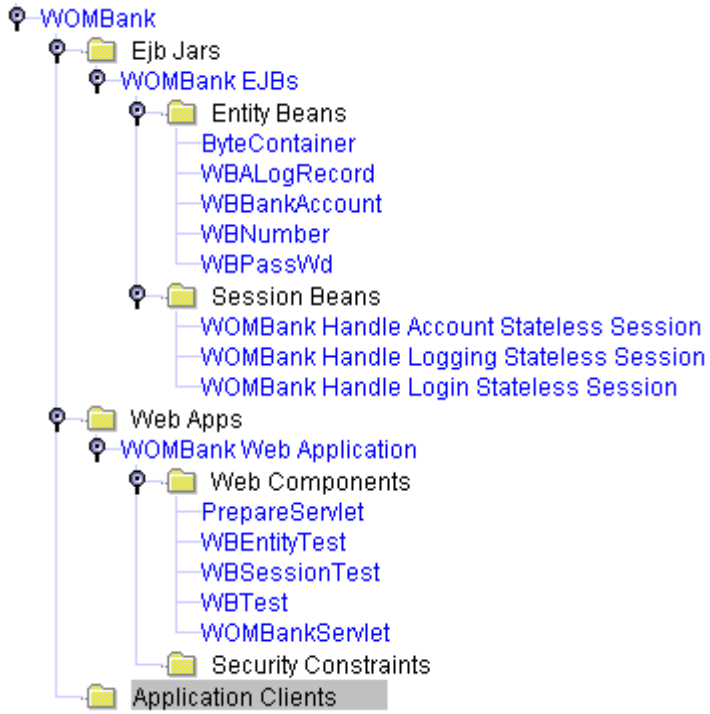


Abbildung 6.3 : Komponenten der WOMBank – Applikation unter Verwendung von EJBs

Des Weiteren wurde der Anwendung eine umfangreiche Java – Dokumentation hinzugefügt, in der die in der gesamten Applikation vorkommenden Klassen und Methoden übersichtlich beschrieben werden. Dazu wurden bereits in die Java – Dateien der Beans und Servlets Kommentare eingefügt, welche dann mit Hilfe des *javadoc* – Befehls ausgewertet werden. Auf die Benutzung dieser Übersicht wird im Kapitel „Vorbereitungen, Installation und Test der WOMBank – Anwendung“ näher eingegangen.

Auch wenn sich dieser Ansatz in der Umsetzung wesentlich von der servletbasierten WOMBank – Anwendung unterscheidet, würde man auch hier von einer 3 – tier – Architektur sprechen. Allerdings muss man die Untergliederung etwas feiner gestalten. Die Präsentation der Anwendung findet wieder auf dem Clientrechner mit Hilfe des Webbrowsers statt. Das WOMBankServlet bildet die Schnittstelle zwischen dem Nutzer und der Geschäftslogik. Für die Datenhaltung wird wieder die DB2 – Datenbank verwendet. Der WebSphere Application Server ist als PlugIn in dem HttpServer verfügbar und Aufrufe von Enterprise Java Beans werden an diesen weitergeleitet. Man könnte auch von einer 4 – tier – Architektur ausgehen, wenn man den WebSphere Application Server als eine eigene Schicht darstellt.

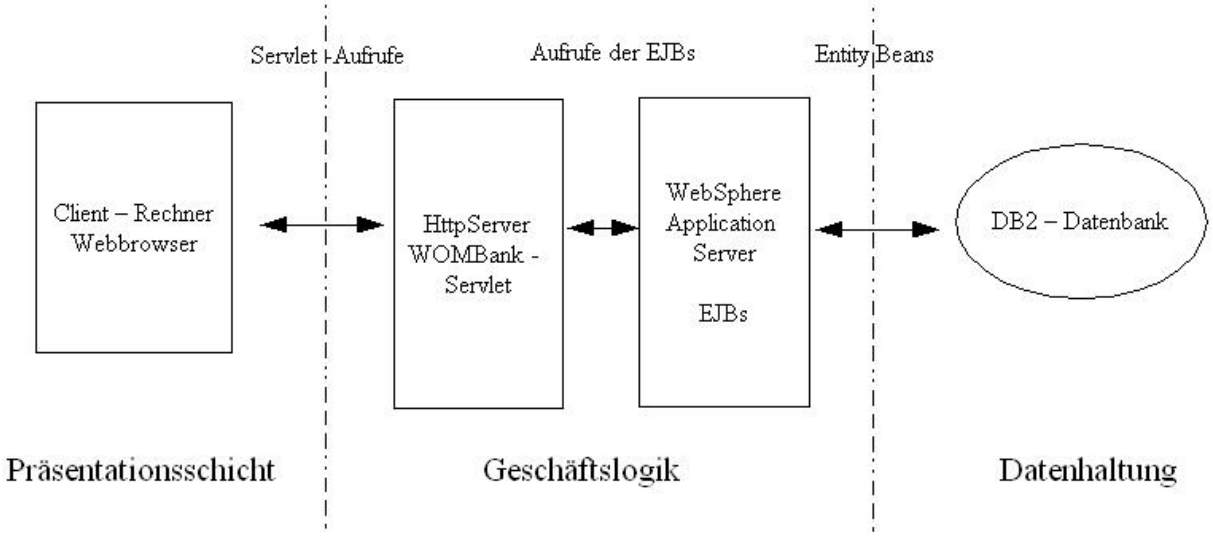


Abbildung 6.4 : 3 – tier – Architektur der WOMBank – Anwendung mit EJB - Ansatz

6.2 Auflistung und Beschreibung der einzelnen Komponenten

Um die Funktionsweise und die genauen Abläufe während einer Anfrage an die Anwendung

besser verstehen zu können, werden im Folgenden die einzelnen Komponenten aufgelistet und genau beschrieben. In Abbildung 6.5 wird der Zusammenhang aller Enterprise Beans der WOMBank – Anwendung und deren Abhängigkeit voneinander dargestellt. Zuerst wird auf die Entity Beans eingegangen, da eine Beschreibung der Funktionsweise, aufgrund der geringen Komplexität der Methoden, sehr einfach ist. Die Auslagerung der etwas komplexeren Geschäftslogik in die Session Beans macht eine Erklärung etwas aufwendiger und da diese auf den Entity Beans aufbauen, erfolgt dies als zweites. Zu guter Letzt wird das Herzstück der WOMBank – Anwendung beschrieben, das WOMBankServlet.

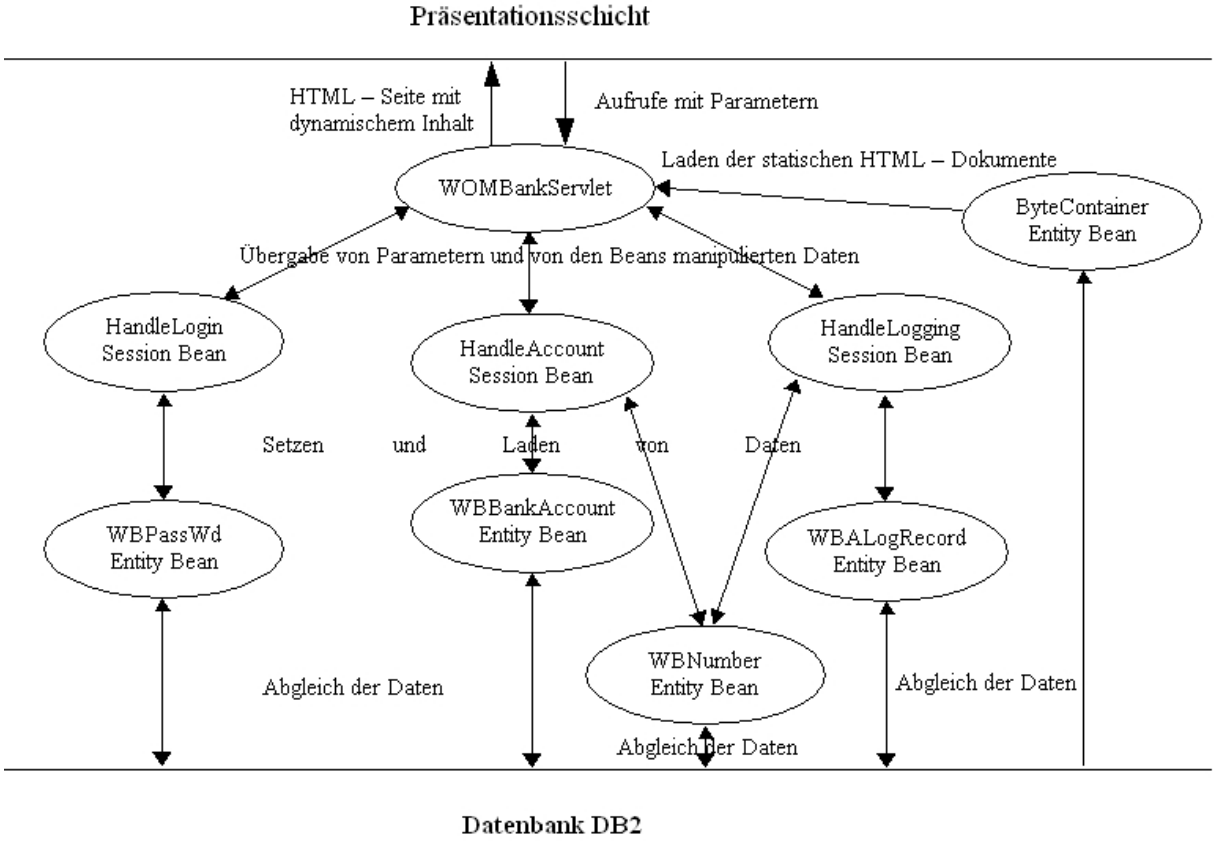


Abbildung 6.5 : Zusammenhang der Enterprise Java Beans der WOMBank – Anwendung

6.2.1 Entity Beans

Entity Beans sind ein einfaches Werkzeug, um die persistente Speicherung von Daten zu organisieren. Dabei wird nicht mehr die Datenbank direkt angesprochen, wie das zum Beispiel bei dem servletbasierten WOMBank – Ansatz der Fall ist, sondern über die Entity Beans.

Diese bestehen aus einem Home – Interface, welches Methoden zum Erstellen als auch zum Suchen einer Instanz zur Verfügung stellt, einem Client – Interface, welches es ermöglicht, Methoden zum Setzen und Laden von Daten aufzurufen und einer Beanklasse, die die eigentliche Implementierung der Bean darstellt. Es ist auch möglich, dass noch eine zusätzliche Klasse, der Primary Key, Bestandteil der Bean ist. Diese dient dazu, mehrere Variablen zu einem Primary Key zusammenzufassen, damit die Einzigartigkeit eines Keys garantiert ist. Das ist notwendig, damit Datensätze, die in der Datenbank gespeichert sind, auch eindeutig wiedergefunden werden können. Bei den folgenden Beschreibungen wird auf jede Komponente der Entity Beans eingegangen, die Bestandteil der WOMBank – Anwendung sind.

6.2.1.1 WBPassWd

Ein Nutzer, der sich für die Verwendung der WOMBank – Anwendung registrieren läßt, wird zur Eingabe eines Usernamens und eines Passwortes aufgefordert. Diese Entity Bean ist für die persistente Speicherung dieser Daten zuständig. Die WBPassWd – Bean gleicht die Daten mit einer einfachen Tabelle in der DB2 ab, die den selben Namen wie die Bean trägt. Sie hat zwei Spalten: *Username* und *Password*. Dabei fungiert der Username als Primary Key, über den der Datensatz wieder in der Datenbank gefunden werden soll. Zwei Datensätze mit demselben Key sind also nicht möglich.
Das Home – Interface umfasst nur die zwei üblichen Methoden. Zum einen die *create* – Methode, welche aufgerufen wird, um eine neue Instanz der Bean und damit einen neuen Datensatz zu erzeugen und zum anderen eine *findByPrimaryKey* – Methode, mit deren Hilfe Daten in der Datenbank gesucht werden. Falls diese nicht zu finden sind, wird eine sogenannte FinderException geworfen.

```
WBPassWd create (String username,String password)
    throws CreateException, RemoteException;

WBPassWd findByPrimaryKey (WBPassWdPK username)
    throws RemoteException, FinderException;
```

Beide Methoden haben das Client – Interface als Rückgabewert, durch das man auf die get – und set – Methoden Zugriff hat, die die Manipulation der Daten ermöglichen. Das sind zum Beispiel *setUsername* zum Speichern des Namens und *getUsername* zum Laden desselbigen aus der Datenbank.

Die Beanklasse enthält unter anderem die programmiertechnische Umsetzung der Methoden, die über die Schnittstellen aufgerufen werden können. Da die Entity Bean vom Container gemanaged wird, ist für einen Großteil der Methoden keine Implementierung notwendig. Das sind vor allem jene, die den Lebenszyklus betreffen. In Kapitel 3.3.7.1 wird auf diese näher eingegangen.

```
public void ejbPostCreate (String username,String password) {}
public void setEntityContext (EntityContext ctx) {}
public void unsetEntityContext() {}
public void ejbLoad() {}
public void ejbStore() {}
public void ejbRemove () {}
public void ejbActivate () {}
public void ejbPassivate () {}
```

Die Implementierung der anderen Methoden ist auch nicht wirklich schwierig. Dabei müssen nur die übergebenen Parameter den Variablen in der Beanklasse zugewiesen werden, alles andere übernimmt der Beancontainer. Das sieht ungefähr so aus:

```
public WBPassWdPK ejbCreate (String username,String password)
        throws CreateException {
        this.username = username;
        this.password = password;
        return null;
}
```

Wie man an den Parametern der *create* – Methode erkennen kann, sind beide Variablen des Typs String. Da es in der Datenbank einen solchen Variablentyp nicht gibt, wird eine Konvertierung zu dem Datentyp Varchar vorgenommen, der im Allgemeinen für die persistente Speicherung von Strings verwendet wird.

6.2.1.2 WBBankAccount

Um eine einfache Version eines Bankkontos zu verwirklichen, müssen einige Daten in einer Datenbank vorgehalten werden. Das sind zum Beispiel der Name des Eigentümers, die Art des Kontos, der aktuelle Kontostand und die Kontonummer. Die Speicherung dieser Daten übernimmt eine Tabelle in der DB2 – Datenbank mit dem Namen WBBankAccount. Den Abgleich der Daten mit dieser Tabelle und die Organisation der Zugriffe übernimmt die WBBankAccount – Entity Bean. Die Tabelle hat vier Spalten mit unterschiedlichen

Datentypen. Das sind *Username*, *Type*, *Number* und *Balance*, welche die oben genannten Daten wiederspiegeln. Als eindeutigen Key reicht hier nicht mehr der Username aus, da ein Nutzer auch mehrere Konten besitzen darf. Es muss also ein Primary Key verwendet werden, der aus dem Usernamen und dem Type zusammengesetzt ist. Dazu ist es notwendig, eine eigene Key – Klasse für die Bean zu schreiben. Neben den besagten Variablen und dem Konstruktor, muss diese außerdem noch zwei Methoden beinhalten, die es erlauben, zwei Primärschlüssel miteinander zu vergleichen und dem Key einen Hashcode zuzuweisen. Die folgenden zwei Methoden realisieren genau dies.

```
public int hashCode() { return (username+type).hashCode(); }

public boolean equals(Object other) {
      if ((other == null) || !(other instanceof WBBankAccountPK)) { return false;}
      WBBankAccountPK otherPK = (WBBankAccountPK) other;
      return username.equals(otherPK.username) &&
            type.equals(otherPK.type);
   }
```

Die Methoden des Home – Interfaces sind denen der vorher beschriebenen Bean ähnlich. Der Unterschied besteht lediglich darin, dass die create – Methode andere Parameter übergeben bekommt.

```
WBBankAccount create (String username,String type,int number,float balance)
      throws CreateException, RemoteException;
```

Auch das Client – Interface ist ähnlich dem der WBPassWd – Bean aufgebaut und beinhaltet aufgrund der größeren Anzahl von Variablen lediglich mehr set – und get – Methoden.
Die Implementierung aller Methoden erfolgt auch in gleicher Weise wie schon bei der vorher erwähnten Entity Bean.

6.2.1.3 WBALogRecord

Um später noch nachvollziehen zu können, welche Aktionen ein Nutzer durchgeführt hat und ob es Fehlermeldungen gab, werden Logbücher in der Datenbank festgehalten. Zum einen ist es wichtig zu wissen, welcher User etwas verändert hat. Weiterhin wird eine Nummer des Eintrages, die Beschreibung der Aktion, das Datum und die genaue Uhrzeit in der

Datenbank vorgehalten. Dabei werden die Daten in die 5 – spaltige Tabelle WBALogRecord abgespeichert. Die Namen der Spalten sind *Username* , *Number* , *Trans*, *Longdate* und *Proctime*. In diesem Fall bilden der Username des Nutzers und die Kontonummer, also *Number*, den Primärschlüssel. Es war also wieder notwendig, eine eigene Keyklasse für die Bean zu schreiben, welche aber große Ähnlichkeit mit der von der WBBankAccount – Bean aufweist.
Zu den Home – und Client – Schnittstellen gibt es nicht viel zu sagen, außer das im Home – Interface eine Findermethode hinzugefügt wurde, die weitere Erklärungen bedarf. Das ist:

```
Collection findByUsername (String username)
        throws RemoteException, FinderException;
```

Mit dieser *findByUsername* – Methode werden alle Datensätze, die zu einem Nutzer gehören, gesucht und in einer Collection, also einer Sammlung dieser Daten, zurückgegeben. Diese Methode wird nicht vollständig durch den Container gestellt, sondern es sind noch einige wenige Schritte nötig, damit sie funktioniert. Zum einen muss eine Klasse implementiert werden, die die *where* – Klausel der SQL – Anweisung enthält und zum anderen muss in einem Deployment Deskriptor festgehalten werden, dass es sich dabei um eine selbst definierte Findermethode handelt. Eine solche Klasse sieht z. Bsp. so aus:

```
public interface WBALogRecordBeanFinderHelper {
   String findByUsernameWhereClause = "username = ?";
}
```

In der Regel wird durch das Assembly Tool ein entsprechender Eintrag in dem *ibm-ejb-jar-ext* – Deskriptor erzeugt. Der Vollständigkeit halber wird dies noch aufgezeigt. Neben den üblichen Angaben zu einer Bean ist dann noch folgende Passage vorzufinden.

```
<finderDescriptors xmi:type="ejbext:WhereClauseFinderDescriptor"
      xmi:id="WhereClauseFinderDescriptor_1" whereClause="username = ?">
     <finderMethodElements xmi:id="MethodElement_1" name="findByUsername"
           parms="java.lang.String" type="Home" description="Created by uMOF to MOF
converter">
     <enterpriseBean xmi:type="ejb:ContainerManagedEntity" href="META-INF/
           ejbjar.xml#WBALogRecord"/>
     </finderMethodElements>
   </finderDescriptors>
```

6.2.1.4 WBNumber

Um einen fortlaufenden Zähler in einer Datenbank zu realisieren, wird diese Entity Bean benötigt. Dabei findet ein Datenabgleich mit der WBNumber – Tabelle statt, die aus zwei Spalten besteht. Der Name der Klasse, die eine fortlaufende Nummerierung verwendet, und der Wert des Zählers selbst. So ist es zum Beispiel möglich jedem Konto eine andere Kontonummer zuzuweisen. Ein Aufruf dieser Bean und deren Funktionalität erfolgt in den Session Beans und wird in diesen Abschnitten näher erläutert.

6.2.1.5 ByteContainer

In der WOMBank – Anwendung werden die zum reibungslosen Ablauf benötigten HTML – Seiten in der Datenbank vorgehalten. Die Organisation dieser Daten übernimmt die ByteContainer – Bean. Dabei wird auf eine Tabelle mit zwei Spalten zugegriffen. Die Spalte *Name* stellt dabei den Dateinamen dar und dient gleichzeitig als Primärschlüssel, *Bytes* ist der eigentliche Inhalt der Datei, der als LongVarchar – Datentyp in der DB2 abgelegt wird. Da Varchar nur maximal 250 Zeichen aufnehmen und eine Datei aber wesentlich größer sein kann, wird dieser Datentyp verwendet. Bevor die WOMBank nach der Installation das erste Mal aufgerufen wird, muss das PrepareServlet aufgerufen werden, um die HTML – Dokumente in die Datenbank zu laden. Eine gründlichere Abhandlung des ByteContainers und des PrepareServlets erfolgt in späteren Abschnitten.

6.2.2 Session Beans

Session Beans ähneln in der Funktionsweise und dem Aufbau den Entity Beans, nur mit dem Unterschied, dass sie keine Informationen in einer Datenbank speichern. Dabei unterscheidet man zwei Arten, stateful und stateless Session Beans. Während die erste Variante zwischen zwei Aufrufen Daten zwischenspeichern kann, wird die zweite vor einem Aufruf immer in den Ausgangszustand versetzt.
Bei der ejb – basierten WOMBank – Anwendung wurde die Implementierung der Geschäftslogik in stateless Session Beans ausgelagert. Das sind vor allem Operationen, die das Login, das Bankkontenmanagement und das Logging betreffen. Somit sind auch drei Beans entstanden, die jeweils einen dieser Bereiche abdecken. Dabei werden in den Methoden häufig nur die Nutzereingaben auf Fehler überprüft und es wird entsprechend darauf reagiert.

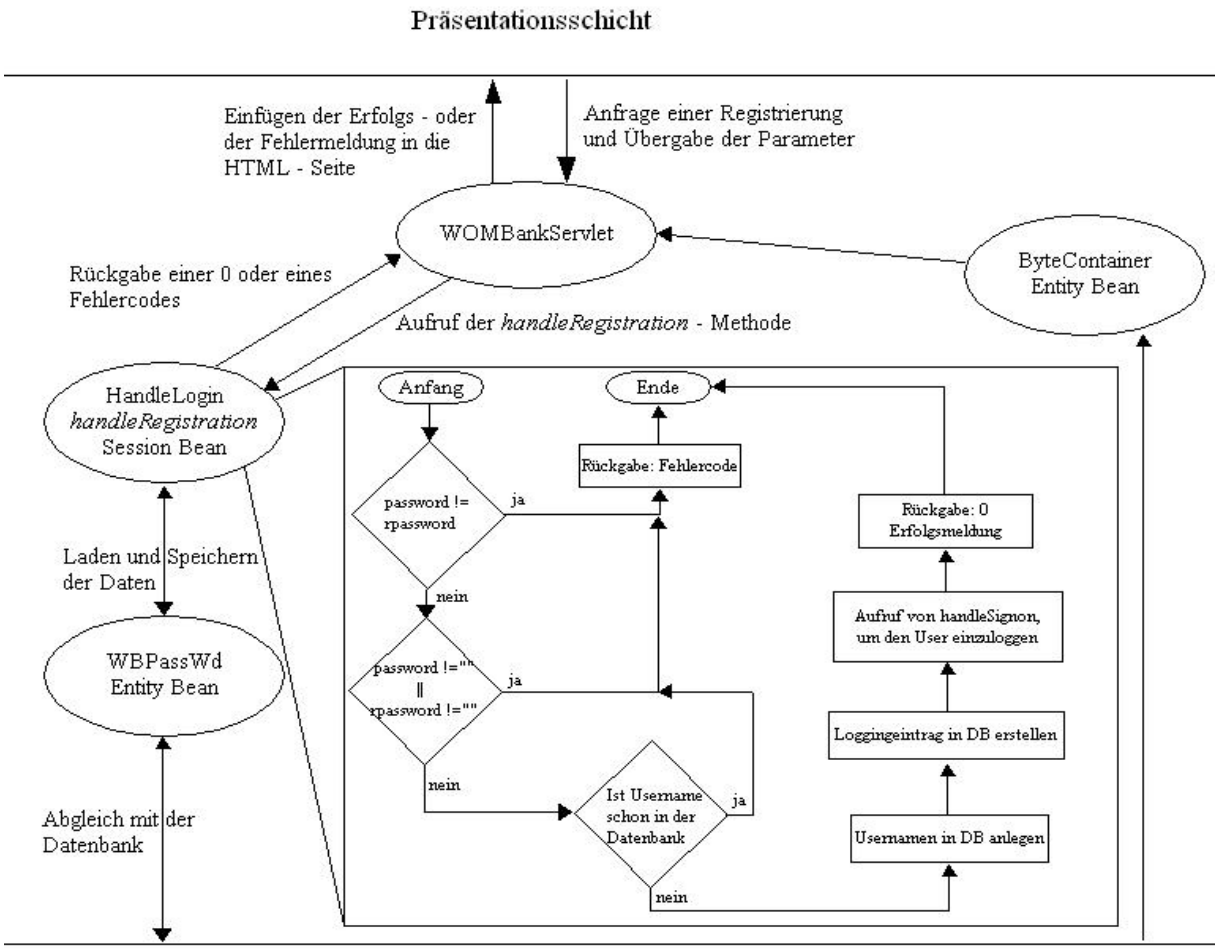
6.2.2.1 HandleLogin

Um die Funktionen der WOMBank nutzen zu können, ist es erforderlich, sich zu registrieren. Das bedeutet, man wird um die Eingabe eines Usernamens und eines Passwortes gebeten. Mit dieser Kombination loggt man sicher später ein und hat dann verschiedene Aktionen zur Auswahl. Die HandleLogin – Session Bean muss zwei verschiedene Vorgänge abdecken, die Registrierung und das Login. Dafür stehen zwei verschiedene Methoden zur Verfügung, *handleRegistration*() und *handleSignon*(), die im Folgenden genau erläutert werden.

```
int handleRegistration(String username,String password,String rpassword) throws
RemoteException;
```

Bei der Registrierung werden drei Parameter benötigt, der Username, das Passwort und eine Wiederholung des Passwortes. Mit diesen Angaben wird unter anderem überprüft, ob sich der Nutzer bei der Eingabe nicht geirrt hat. Sollten *password* und *rpassword* nicht genau übereinstimmen, wird ein Fehlercode an das aufrufende Objekt zurückgeschickt. Dafür dient der Rückgabewert der Funktion. Liegt dieser unter 0, ist bei dem Vorgang etwas schief gelaufen. Im WOMBankServlet erfolgt eine Übersetzung des Fehlercodes, so dass dem Nutzer ein aussagekräftiger Text ausgegeben werden kann. In der folgenden Abbildung ist der Ablauf einer Registrierung dargestellt.

Abbildung 6.6 : Ablauf einer Registrierung

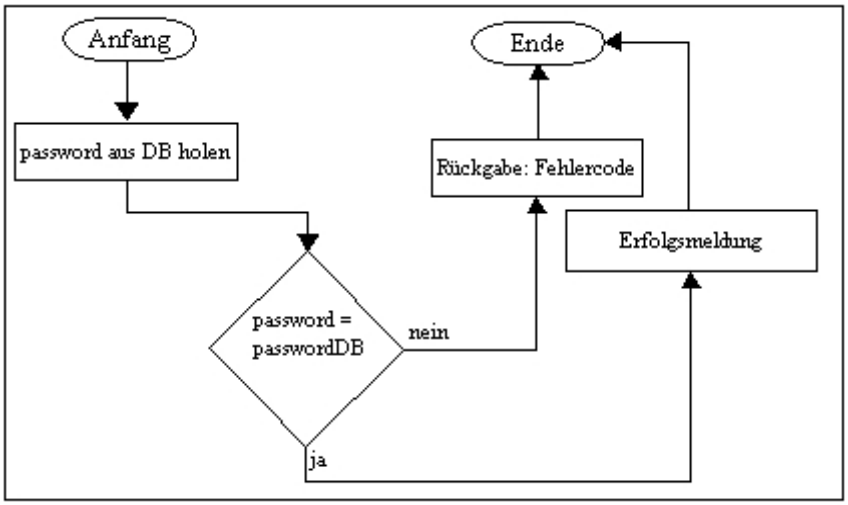


Nach der erfolgreichen Registrierung wird die *handleSignon* – Methode aufgerufen, um den Nutzer einzuloggen.

```
int handleSignOn(String username,String password) throws RemoteException;
```

Mit Hilfe dieser Funktion wird das eingegebene Passwort mit dem aus der Datenbank verglichen. Bei Gleichheit wird eine Erfolgsmeldung, anderenfalls eine Fehlermeldung an das WOMBankServlet gesendet. Der eigentliche Einloggvorgang findet im WOMBankServlet statt.

Abbildung 6.7 : handleSignon - Methode



6.2.2.2 HandleAccount

Für die Organisation des Bankkontos, also das Abheben und Einzahlen von „Geld“, das Erstellen eines Kontos und das Transferieren von „Geld“ auf ein anderes Konto, ist die HandleAccount – Session Bean verantwortlich. Um ein neues Bankkonto für einen bestimmten User zu eröffnen, wird folgende Methode verwendet:

```
int createAccount(String username,String type,float balance)
```

Als Parameter werden der Username, der eingezahlte Geldbetrag und der Typ des Kontos übergeben. Die WOMBank – Anwendung kennt vier verschiedene Arten von Konten, die unter folgenden Bezeichnungen geführt werden, Checking, Savings, Holiday und Special. Beim Erstellen muss sich der User für einen Kontotypen entscheiden. In der *createAccount* – Methode werden die Angaben des Users auf Fehler überprüft. So werden negative Eingaben bei den Geldbeträgen während der Kontoeröffnung abgefangen. Weiterhin ist es nicht gestattet, das ein Kunde mehr als ein Konto desselben Typs besitzt. Sind alle Angaben korrekt, wird mit den entsprechenden Entity Beans ein neues Konto und ein Loggingeintrag in der Datenbank erzeugt.
Um eine fortlaufende Kontonummerierung zu gewährleisten, wird die nächste zu verwendende Kontonummer in der Datenbank gespeichert und kann bei Bedarf mit Hilfe der *getNextNumber* – Methode abgerufen werden. Dabei wird der Datensatz mit dem Primärschlüssel „bankaccount“ über die WBNumber – Entity Bean geladen. Danach wird die Nummer ausgelesen und anschließend, um eins erhöht, wieder in die Datenbank

geschrieben.

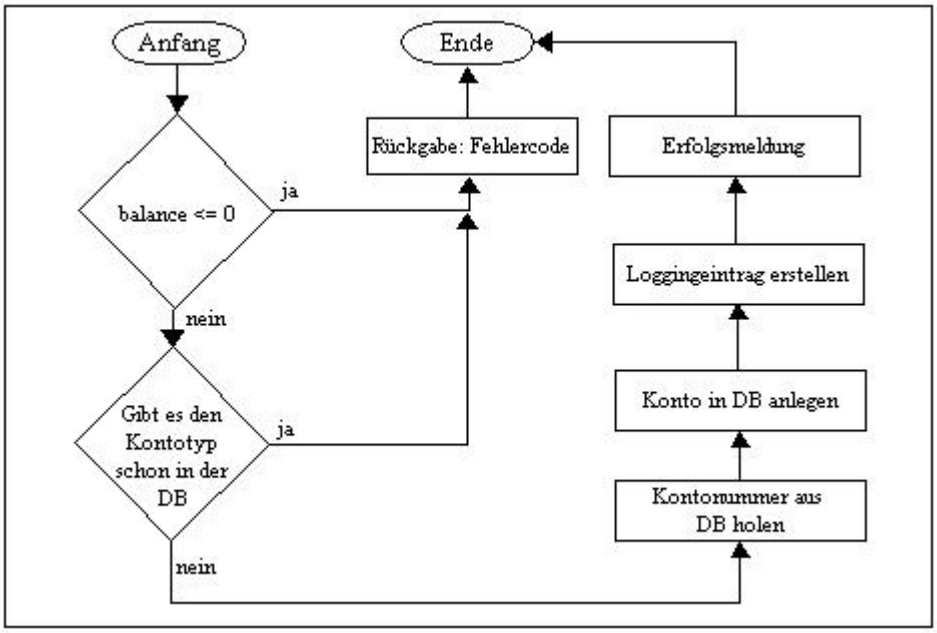


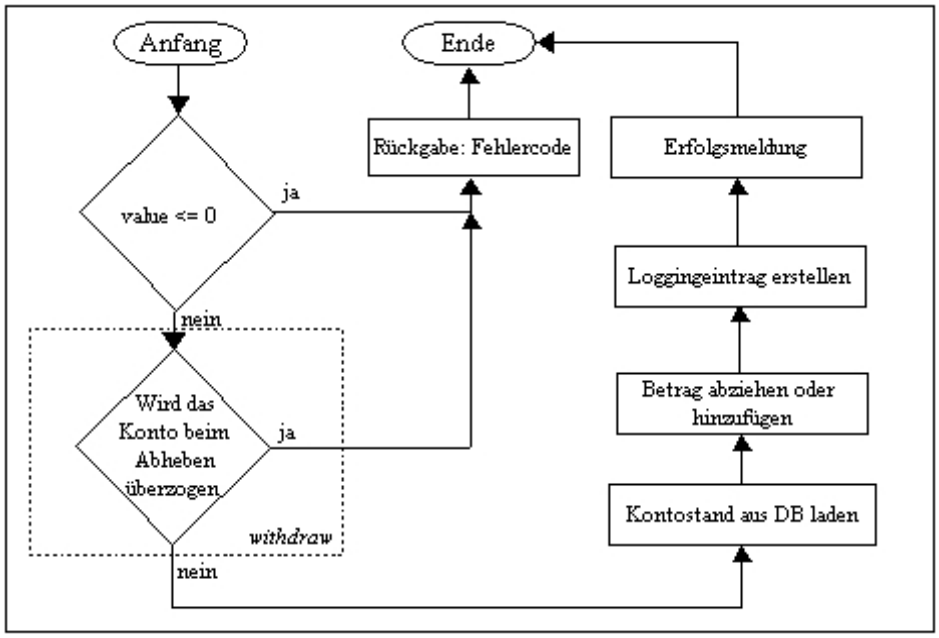
Abbildung 6.8 : Ablauf einer Kontoeröffnung in der WOMBank

Kunden der WOMBank haben natürlich auch die Möglichkeit, Geld auf eines ihrer Konten einzuzahlen oder welches abzuheben. Diese beiden Vorgänge werden über die Methoden *withdraw* und *deposit* realisiert. Da sich der Aufbau der zwei Funktionen ähnelt, werden sie im Folgenden bei der Beschreibung zusammengefasst.

```
int withdraw(String username,String type,float value)
int deposit(String username,String type,float value)
```

Als Parameter werden der Username des Kontobesitzers, der Typ des Kontos und der abzuhebende beziehungsweise auszuzahlende Geldbetrag übergeben. Mit den ersten beiden kann das zu bearbeitende Konto eindeutig bestimmt werden. Im Falle des Einzahlens von Geld wird nun der Betrag dem Konto gut geschrieben und eine Erfolgsmeldung an das WOMBankServlet gesendet. Beim Abheben allerdings muss vor der Auszahlung noch überprüft werden, ob das Konto dabei überzogen wird. Ist dem nicht so, wird das Geld vom Konto abgehoben. Auch nach erfolgreichem Abschluss dieser Aktion wird das Servlet benachrichtigt. Die folgende Abbildung zeigt noch einmal den Ablaufplan der beiden Vorgänge:

Abbildung 6.9 : Ablauf einer Aus – bzw. Einzahlung



Eine weitere Funktion, die von der Onlinebank zur Verfügung gestellt wird, ist der Transfer von Geld zwischen den eigenen Konten. Realisiert wird dieser Vorgang über die *transfer* – Methode:

```
int transfer(String username,String type1,String type2,float value)
```

Parameter dieser Funktion der Username des WOMBank – Nutzers, die beiden Kontotypen, zwischen denen eine Überweisung stattfinden soll und der zu transferierende Betrag. Letztendlich unterscheidet sich die Vorgehensweise nicht großartig von den anderen Methoden, allerdings muss hierzu noch eine Bemerkung gemacht werden. Bei diesem Vorgang muss ein Geldbetrag von einem Konto abgehoben und auf das andere Konto wieder eingezahlt werden. Sollte dazwischen ein Fehler auftreten, ist es nicht mehr nachvollziehbar, inwieweit der Transfer schon abgeschlossen war. Auch die Logging – Dateien können in diesem Fall nicht zu Rate gezogen, da dort genau dasselbe Problem besteht. Die Loggingdatensätze werden erst geschrieben, nachdem die eigentliche Aufgabe schon erledigt ist. Man müßte da auf eine Atomizität von Vorgängen bestehen, die nicht mit einfachen Mitteln umsetzbar wäre, wenn dies überhaupt mit der Java – Programmiersprache möglich ist. Für eine richtige Bank wäre solch ein Zustand natürlich nicht denkbar, allerdings soll die WOMBank – Applikation nur die Fähigkeiten von Enterprise Java Beans und die einfache Umsetzung von komplexen Modellen aufzeigen. Eine detailgetreue

Implementierung eines Onlinebanken – Konzeptes, was Sicherheit, Datenschutz und Verfügbarkeit angeht, würde den Rahmen jeder Diplomarbeit sprengen.

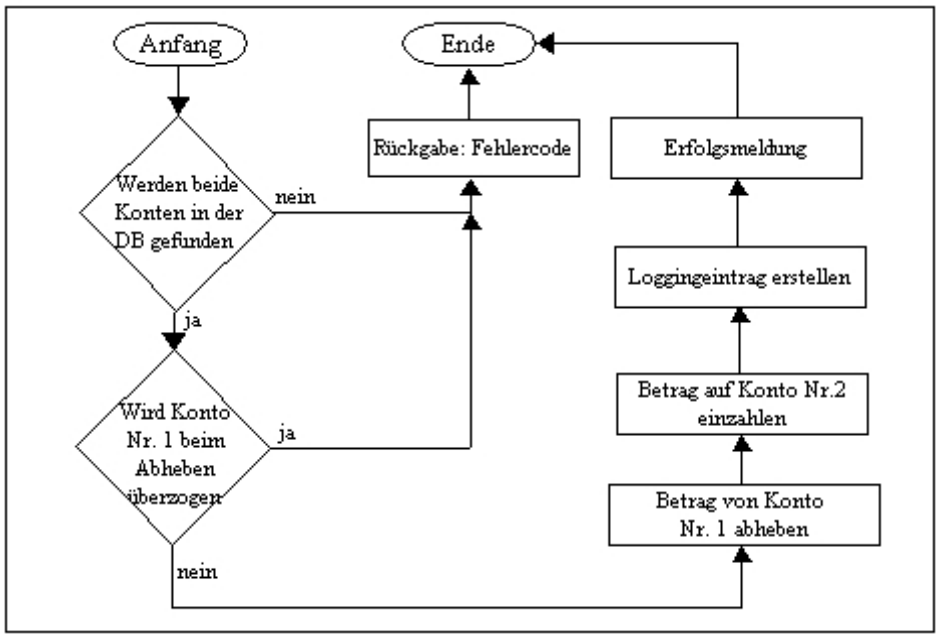


Abbildung 6.10 : Ablauf der *transfer* - Methode

Es gibt noch eine weitere Funktion, die Bestandteil der HandleAccount – Session Bean ist.

String getBalance(String username,String type)

Mit dieser *getBalance* – Methode wird lediglich von einem, durch die Parameter angegebenes, Konto der Kontostand abgerufen.

6.2.2.3 HandleLogging

In den beiden zuvor abgehandelten Session Beans wird häufig davon Gebrauch gemacht, einen Loggingeintrag für die Aktivitäten von Nutzern in der Datenbank zu erstellen. Dabei erhält die handleLogging – Methode bereits alle nötigen Informationen über die Parameter.

int handleLogging(String username,String trans,Date longdate,Timestamp proctime)

Beim Aufruf wird in der Datenbank mit Hilfe der WBALogRecord – Entity Bean ein Datensatz erzeugt. Die übergebenen Variablen repräsentieren dabei folgende Daten. Den genauen Wortlaut der Transaktion stellt *trans* dar, *longdate* das Datum und *proctime* die Zeit. Auch

die Loggingeinträge werden genau wie die Konten durchnummeriert. Dazu kommt wieder die *getNextNumber* – Methode zum Einsatz, der diesmal "alogrecord" als Primärschlüssel übergeben wird. Damit wird die nächste zu verwendende Nummerierung für einen Logbucheintrag aus der Datenbank ausgelesen, um eins erhöht und wieder hineingeschrieben.

Bei der Beschreibung der WBALogRecord – Entity Bean wurde auf eine *findByUsername* – Methode besonders eingegangen. Damit können alle Loggingeinträge aus der Datenbank geladen werden, die zu einem bestimmten Nutzer gehören. Die gesammelten Daten werden dann in einer java.util.Collection zurückgeliefert, die dann noch aufbereitet werden müssen, um sie im Browser anzeigen zu können. Diese Aufgabe übernimmt die *getLog* – Methode.

```
String getLog(String username)
```

Der Rückgabewert dieser Funktion ist ein String der folgende Form hat:

```
Datum  Uhrzeit  Wortlaut der Transaktion
Datum  Uhrzeit  Wortlaut der Transaktion
...
```

6.2.3 WOMBankServlet

Das Kernstück der WOMBank – Applikation ist das WOMBankServlet. Bei einer Anfrage des Nutzers an das Servlet wird aufgrund der übergebenen Parameter und dem Zustand der vom Servlet gespeicherten Variablen entschieden, welche Aufrufe an die Beans getätigt werden. Ausschlaggebend dafür ist der *action* – Parameter. Mit ihm wird klar festgelegt, welche Aufgaben im momentanen Schritt auszuführen sind. Dabei werden sechs verschiedene Werte akzeptiert.

action=register	Ein Kunde möchte sich registrieren
action=signon	Der User loggt sich ein
action=new	Ein eingeloggter Nutzer möchte ein neues Konto eröffnen
action=exist	Ein eingeloggter Kunde ruft seine Kontenübersicht ab
action=tlog	Ein eingeloggter User ruft die Logdaten seiner letzten Aktionen auf

action=register	Ein Kunde möchte sich registrieren
action=logout	Der Nutzer loggt sich aus
action=info	Der Nutzer möchte sich die Informationsseite der WOMBank ansehen

Alle anderen Werte führen zu einer Fehlermeldung.
Wenn sich ein Kunde für die Nutzung der WOMBank registrieren möchte, es wird also der *action* – Parameter mit dem Wert *register* an das WOMBankServlet übergeben, wird aus der Datenbank die **register.html** mit Hilfe der *getFileContent* – Methode geladen, um dem User mittels einer Eingabemaske die Möglichkeit zu bieten, seine Nutzerdaten einzutragen. Dabei spricht die getFileContent – Funktion die ByteContainer – Bean an und übergibt als Primärschlüssel den Dateinamen. Der Rückgabewert ist der Inhalt der Datei in Form eines Strings. Für den Fall, das der User bei der Eingabe keinen Fehler gemacht hat und die Registrierung erfolgreich verlaufen ist, das heißt die *handleRegistration* – Methode der HandleLogin – Session Bean liefert eine 0 zurück, wird der Nutzer eingeloggt (action=signon).
Dieser Mechanismus ist ziemlich einfach gehalten und soll keineswegs Sicherheitsüberprüfungen standhalten. Dabei wird lediglich der Username des Nutzers, in der aktuellen Sitzung (Session) gespeichert. Das geschieht mit folgendem Programmfragment:

```
if(success) session.setAttribute("uname",username);
```

Bei jeder Aktion, die einen eingeloggten User voraussetzt, wird überprüft, ob sich ein Usernamen in der Session befindet. Wenn ja, wird mit diesem gearbeitet, ansonsten wird der Kunde gebeten, sich einzuloggen.
Beim Ausloggen (action=logout) wird einfach der Username aus der Session entfernt und diese für ungültig erklärt.

```
session.removeAttribute("uname");
session.invalidate();
```

Bei der nächsten Anfrage eines Nutzers wird eine neue Sitzung angefangen.

Wenn ein Kunde ein neues Konto eröffnen möchte, wird dem Servlet action=new übergeben. Daraufhin wird aus der Datenbank die Datei **new.html** geladen und dem Nutzer wird eine Eingabemaske zur Verfügung gestellt. Dort kann der User Art des Kontos und den

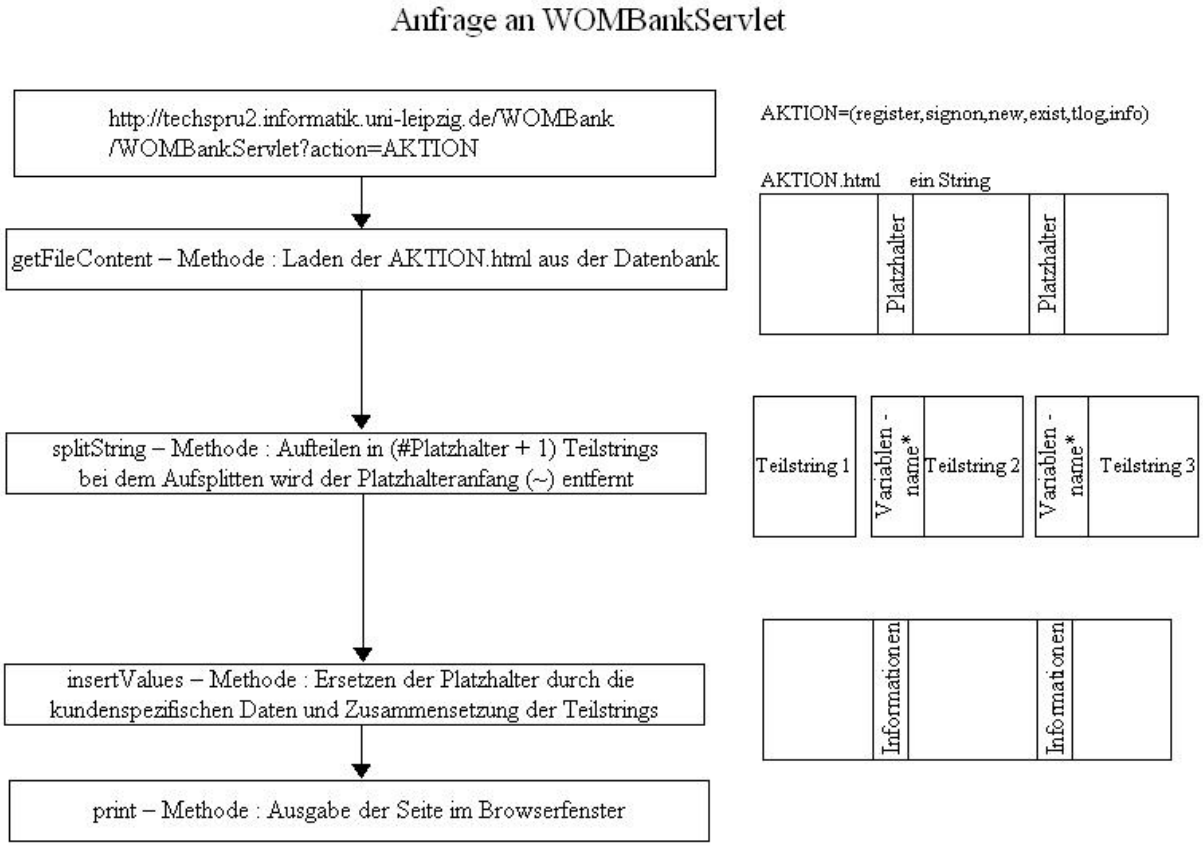
Eröffnungsbetrag eintragen.
Ein Kunde hat auch die Möglichkeit, sich einen Überblick über seine momentane Kontosituation zu verschaffen (action=exist) und auch das Logbuch seiner eigenen Aktionen (action=tlog) einzusehen. Aus der Datenbank wird dann entweder die **exist.html** oder **tlog.html** geladen.

Für eine gewisse Benutzerfreundlichkeit der WOMBank ist es notwendig, das Informationen dynamisch in die Seiten integriert werden. Das sind zum Beispiel die Kontodaten des jeweiligen Nutzers oder Fehlermeldungen. Um das zu realisieren, sind in den statischen HTML – Seiten Platzhalter eingebaut, die mit den jeweiligen Daten ersetzt werden. Dafür müssen sie aber von dem HTML – Code unterschieden werden können, was eine besondere Form erfordert. Der Anfang eines solchen Platzhalters bildet eine Tilde (~) und beendet wird er mit einem Stern (*). Dazwischen steht der eigentliche Variablennamen, dessen Wert den Platz ersetzen soll. Bevor dies geschehen kann, muss vorher der String aufgesplittet werden. Das übernimmt die *splitString* – Methode.

```
String[] splitString(String content,String splitCriteria)
```

Sie bekommt den zu splittenden String und das Kriterium übergeben, nach dem dieser aufgeteilt werden soll. Im Falle der WOMBank wäre das die Tilde (~). Nach dem erfolgreichen Aufsplitten wird ein Array von Teilstrings zurückgegeben. Mit diesen kann nun die *insertValues* – Methode den String wieder zusammenbauen und die kundenspezifischen Kontodaten und Fehlermeldungen einfügen. Der genaue Zusammenhang der einzelnen Funktionen wird in folgender Grafik dargestellt.

Abbildung 6.11 : Einfügen von dynamischen Inhalten in statische HTML - Seiten



7 Vorbereitungen, Installation und Test der WOMBank

Da der Weg vom Erstellen der EAR – Datei, über die Installation bis hin zum erfolgreichen Aufrufen der Anwendung keineswegs ein kurzer ist und auch gewisse Unannehmlichkeiten aufweisen kann, wird diesem Thema ein eigenes Kapitel gewidmet. Die vorgegebenen Vorgehensweisen sind allerdings nur Vorschläge und stellen nicht den einzig möglichen Weg zum Ziel dar.

7.1 Vorbereitungen

Bevor die WOMBank – Applikation installiert werden kann, sollte die neueste Version des WOMBank.ear zur Verfügung stehen. Das Zusammenbauen dieser EAR – Datei wird im folgenden Abschnitt kurz erläutert. Des Weiteren sollten die Tabellen in der Datenbank vorhanden sein, auf die die Entity Beans der Anwendung zugreifen.

7.1.1 Erstellen des WOMBank.ear

Das Erstellen der EAR – Datei läuft weitestgehend automatisch und erfordert keine Eingriffe durch den Nutzer. Vor dem Start müssen lediglich ein paar Voraussetzungen erfüllt sein und Anpassungen an das gerade verwendete System vorgenommen werden.
Die Sourcecode – Dateien und auch schon die fertige Anwendung befinden sich im Ordner **\Diplom Thomas Kumke\Diplomapplikationen** auf der CD, welche der Diplomarbeit beiliegt. Eine genaue Auflistung der Verzeichnisse und Dateien auf der CD ist im Anhang B zu finden. Wenn die Anwendungen nur installiert werden sollen, liegen dafür im Unterverzeichnis **\lib** die schon fertig gebauten Dateien **WOMBank.ear** und **HelloWorld.ear** bereit.
Falls Änderungen an den Quelldateien der Anwendungen vorgenommen werden, müssen die Archive neu erstellt werden. Dazu ist es erforderlich, dass eine aktuelle Version des Java Software Development Kit Standard Edition auf dem System installiert ist. Auf der CD liegt die J2SDK1.4.2_07 – Installationsdatei im Verzeichnis **\Diplom Thomas Kumke\Software\J2SDK** vor. Bei der Installation ist nichts Besonderes zu beachten, nur das das ausgewählte Installationsverzeichnis nachfolgend als JAVA_HOME bezeichnet wird. Des Weiteren muss das gesamte Unterverzeichnis **\Diplomapplikationen** auf die Festplatte kopiert werden. Falls ein Schreibschutz auf alle Dateien besteht, sollte dieser entfernt werden. In dem Unterordner **\src\Diplomarbeit** befindet sich die Stapelverarbeitungsdatei *buildApplication.bat*, die nach Aufruf die EAR – Dateien erstellt. An dieser müssen noch Veränderungen vorgenommen werden, um sich an das vorhandene System anzupassen. Der folgende Ausschnitt zeigt die zu verändernden Zeilen.

```
SET APPLICATION_HOME=D:\Diplomapplikationen
SET JAVA_HOME=D:\j2sdk1.4.0_07

set ANT_ARGS=-Dbasedir="D:\Diplomapplikationen"
```

APPLICATION_HOME stellt das Verzeichnis dar, in der die gesamten Quelldateien vorliegen. Würde der Unterordner **\Diplomapplikationen** also auf Festplatte **C:** kopiert werden, müßte der Wert der Variablen entsprechend auf **C:\Diplomapplikationen** geändert werden. Das Gleiche gilt auch für die basedir – Property, welche in der letzten Zeile des Ausschnitts definiert wird. JAVA_HOME das zuvor erwähnte Installationsverzeichnis des Java SDK.
Nachdem alle Anpassungen vorgenommen wurden, können die Anwendungen neu erstellt

werden. Für das WOMBank.ear ist der Befehl *buildApplication makeWOMBank* in der Eingabeaufforderung einzugeben.

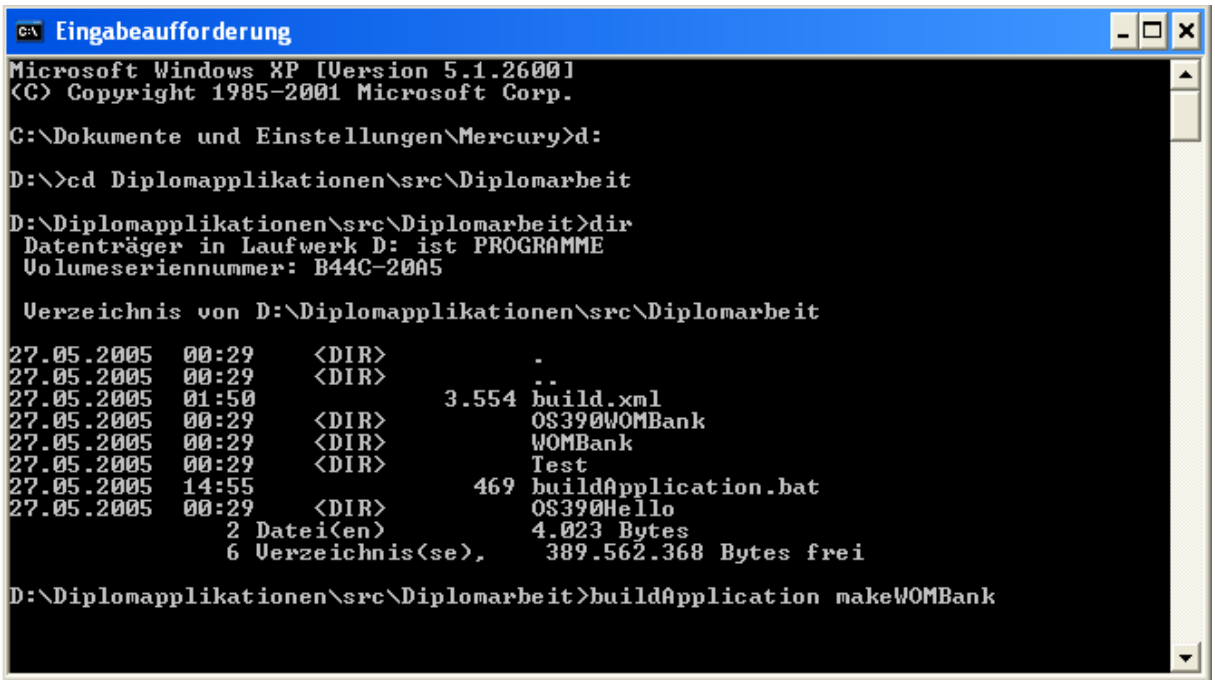


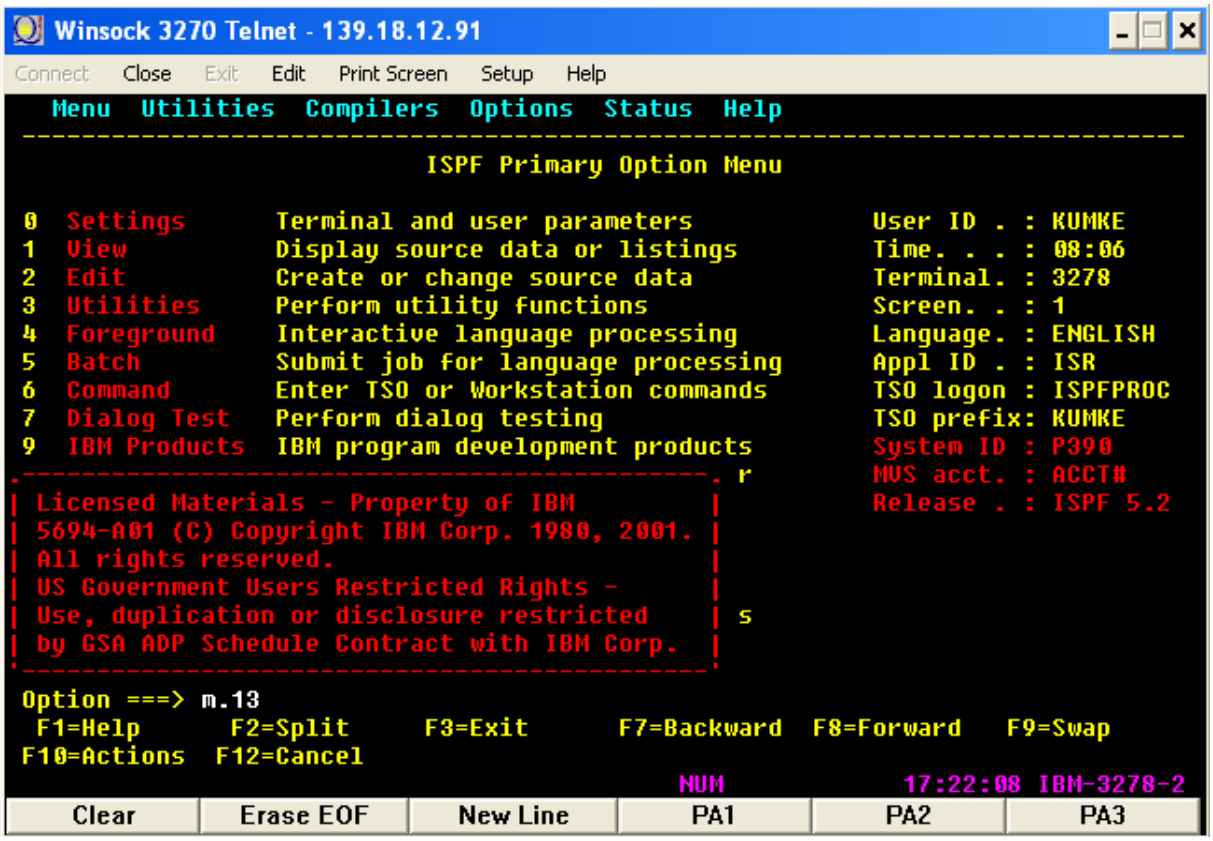
Abbildung 7.1 : Aufruf der Batchdatei zur Erstellung der WOMBank.ear - Datei

7.1.2 Erstellen der Tabellen in der Datenbank

Die WOMBank – Anwendung verwendet Enterprise Java Beans um den Zugriff auf Daten in der Datenbank zu organisieren. Damit dies geschehen kann, müssen die Tabellen vorhanden sein. Auf dem YODA – Rechner sind diese bereits vorhanden. Die Vorgehensweise bei der Erstellung gleicht größtenteils der in **Abschnitt 5.2.4** beschriebenen. Das Werkzeug, welches zur Kommunikation mit dem z/OS – Rechner verwendet wird, ist das QWS3270 – Programm. Eine Installationsversion des Tools befindet sich auf der CD im Verzeichnis

\Diplom Thomas Kumke\Software\QWS3270. Nach dem Starten und Verbinden mit dem YODA – Rechner wird das Administrationsmenu für die DB2 – Datenbank ausgewählt (Menüpunkt 13).

Abbildung 7.2 : Wahl des Administrationsmenüs für die DB2



Um den Eingabebildschirm zu gelangen, wird im folgenden Menu der Punkt 2 und danach der Punkt 1 gewählt. Dort können die Tabellen durch folgende SQL – Befehle erstellt werden. Um Fehler bei der Eingabe besser ausmachen zu können, ist es ratsam, die Tabellen nacheinander und einzeln zu erstellen.

```
create table cbasru2.WBPassWd
   (username varchar(50) not null,
    password varchar(50),
    constraint WBPassWdPK primary key(username))
    in estoredb.estorets; commit;

create unique index cbasru2.WBPassWdix
            on cbasru2.WBPassWd(username asc); commit;

create table cbasru2.WBBankAccount
   (username varchar(50) not null,
    type varchar(10) not null,
    number integer,
    balance varchar(250),
    constraint WBBankAccountPK primary key(username,type))
    in estoredb.estorets; commit;
```

```
create unique index cbasru2.WBBankAccountix
            on cbasru2.WBBankAccount(username asc,type asc); commit;

create table cbasru2.WBALogRecord
   (username varchar(50) not null,
    trans varchar(250),
    number integer not null,
    longdate date,
    proctime timestamp,
    constraint WBALogRecordPK primary key(username,number))
    in estoredb.estorets; commit;

create unique index cbasru2.WBALogRecordix
            on cbasru2.WBAlogRecord(username asc,number asc); commit;

create table cbasru2.WBNumber
   (primkey varchar(250) not null,
    number integer,
    constraint WBNumberPK primary key(primkey))
    in estoredb.estorets; commit;

create unique index cbasru2.WBNumberix
            on cbasru2.WBNumber(primkey asc); commit;

create table cbasru2.ByteContainer
   (name varchar(250) not null,
    bytes long varchar,
    length decimal(19,0),
    constraint ByteContainerPK primary key(name))
    in estoredb.estorets; commit;

create unique index cbasru2.ByteContainerix
            on cbasru2.ByteContainer(name asc); commit;
```

Es besteht auch die Möglichkeit, die SQL – Befehle aus einer Datei einzulesen. Die Dateien *createTable* (zum Erzeugen) und *dropTable* (zum Löschen) befinden sich auf der CD in dem Ordner **\Diplom Thomas Kumke\DB2 Tabellen**.

7.2 Installation

Im folgenden Abschnitt werden alle Schritte beschrieben, die für eine Installation nötig sind.

7.2.1 Installation der Anwendung

Für die Installation der WOMBank – Anwendung wird das **S**ystem **M**anagement **U**ser **I**nterface (kurz SMUI) verwendet. Eine installationsbereite Version des Tools befindet sich auf der CD in dem Ordner **\Diplom Thomas Kumke\Software\WAS-Tools**. Nach der Installation ist es von Vorteil eine Umgebungsvariable anzulegen, um die wiederholte Eingabe der Loginparameter zu vermeiden. Diese Einstellungen werden unter Windows XP bei den Systemeigenschaften vorgenommen.

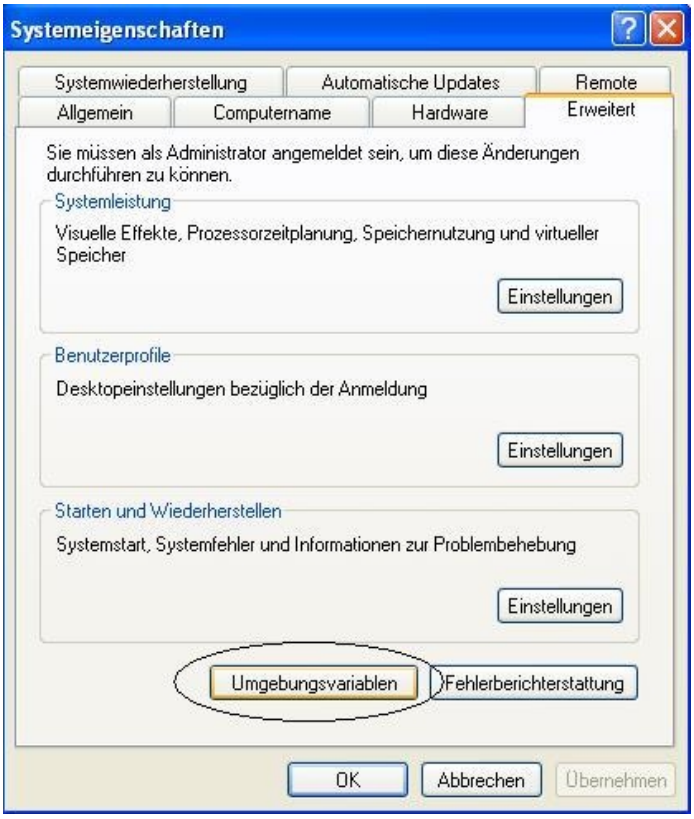
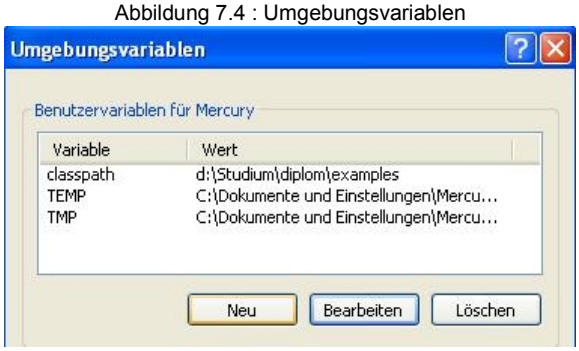


Abbildung 7.3 : Systemeigenschaften unter Windows XP

Nach einem Klick auf den Button *Umgebungsvariablen*, erscheint ein zweites Fenster mit Informationen zu allen System – und nutzerspezifischen Variablen. Unter den Nutzervariablen befindet sich der Button *Neu* zur Definition einer neuen Variable.

Abbildung 7.4 : Umgebungsvariablen



Dort wird eine Variable mit dem Namen BBONPARM und dem Wert *-bootstrapserver 139.18.12.91 -loginuser CBADMIN -loginpassword CBADMIN* definiert. Dabei stellen die Parameter *bootstrapserver* die IP – Adresse des z/OS – Rechners, *loginuser* den Usernamen und *loginpassword* das Passwort für den Administratoraccount dar. Alle loginspezifischen Parameter sind somit beim Start des SMUI verfügbar. Nachdem die Richtigkeit der Angaben bestätigt und überprüft wurde, wird der Startbildschirm sichtbar. Das kann einige Minuten dauern.

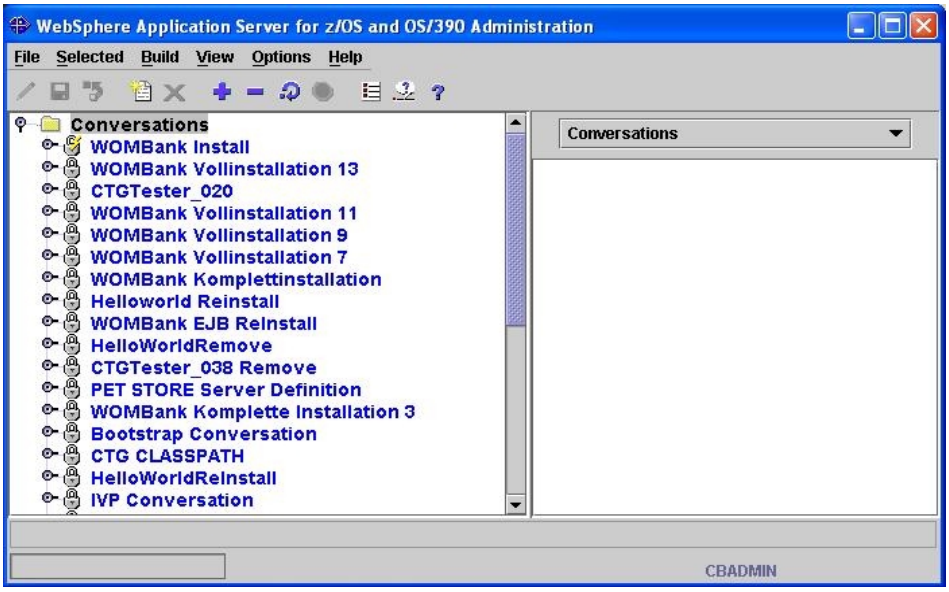


Abbildung 7.5 : Startbildschirm des SMUI

Die Vorgehensweise bei der Installation ist größtenteils die gleiche, wie bei der HelloWorld – Anwendung aus Kapitel 5. Eine installationsbereite Version der Datei WOMBank.ear befindet sich auf der CD im Verzeichnis **\Diplom Thomas Kumke\Diplomapplikationen\lib\Diplomarbeit.** Um diese Datei zu installieren, müssen nur die Anweisungen in Anschnitt 5.2.4 befolgt werden. Das bedeutet also, es wird eine neue *Conversation* erstellt, dann erfolgt durch die Wahl des Menüpunktes *Install J2EE Application* die Installation der Anwendung. Nachdem die *Conversation* mit *commit* bestätigt und anschliessend aktiviert wurde, kann die sie im Browser aufgerufen werden. Weitere Informationen zur Installation einer J2EE – Anwendung werden in den beiden Tutorien geliefert, die diese Diplomarbeit begleiten. Sie sind auf der CD im Ordner **\Diplom Thomas Kumke\Tutorien** zu finden.

Sollte aufgrund von Änderungen im Quelltext eine neue Version der WOMBank.ear – Datei erstellt worden sein, muss diese vor der Installation mit dem SMUI noch mit Hilfe des Assembly Tools darauf vorbereitet werden. Dieser Vorgang wird ebenfalls in Abschnitt 5.2.4 genauer behandelt und kommt auch in den beiden Tutorien zur Sprache. Siehe dazu auch [9].

Bei der Installation muss noch ein weiterer Punkt beachtet werden. Da auf dem z/OS – Rechner der WebSphere Application Server nur als PlugIn für den HttpServer vorliegt und alle Http – Anfragen an diesen gerichtet werden, muss in einer Konfigurationsdatei festgehalten werden, welche Anfragen davon an den WAS weitergeleitet werden sollen. Wird das versäumt, liefert der Server den **Fehler 404 – Seite nicht gefunden** zurück. Die Einstellungen vorzunehmen, ist nicht schwer, vorausgesetzt man besitzt die nötigen Rechte dazu.
Die zu editierende Konfigurationsdatei des HttpServers ist die *httpd.conf.* Eine einfache Möglichkeit, die Einstellungen zu verändern, ist das Downloaden der Datei mit einem FTP – Client (zum Beispiel WS_FTP95), das Verändern der Datei auf dem eigenen System und das Zurücksenden der neuen Datei wieder via FTP.
Eine installationsbereite Version des WS_FTP – Programms liegt auf der CD im Verzeichnis **\Diplom Thomas Kumke\Software\WS_FTP**. Nach der Installation sind noch Einstellungen vorzunehmen, damit keine Probleme bei der Kommunikation mit dem z/OS – Rechner auftreten. Durch das Betätigen des *Connect* – Buttons wird ein Fenster mit verschiedenen Loginprofilen geöffnet. Es sollte ein Neues mit den in Abbildung 7.6 dargestellten Einstellungen angelegt werden. Selbstverständlich sind die Punkte *User ID* und *Password* selbst zu ergänzen.
Ist der Einloggvorgang erfolgreich abgeschlossen, kann man die Datei *httpd.conf* herunterladen. Sie befindet sich im Verzeichnis **\web\httpd1** (das ist eine eins) . Vor dem Downloaden sollte noch

Abbildung 7.6 : Loginprofil für WS_FTP

beachtet werden, dass als Übertragungsmethode ASCII (unter den beiden Dateifenstern) gewählt wurde.

Ist die Übertragung erfolgreich verlaufen, kann die Datei mit einem einfachen Texteditor geöffnet und auch bearbeitet werden. Die zwei hervorgehobenen Zeilen sind der Datei hinzuzufügen, wenn Sie nicht schon vorhanden sind. Da die Datei ziemlich groß, bietet es sich an, nach einem Wort in dem Textausschnitt zu suchen.

```
#   ================================
#   *** WAS 4.0 Directives ***
#   ================================

ServerInit   /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:init_exit /
usr/lpp/WebSphere, /web/httpd1/was.conf
...
Service      /Test/*      /
usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:service_exit
Service      /WOMBank/*      /
usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:service_exit
...
ServerTerm   /usr/lpp/WebSphere/WebServerPlugIn/bin/was400plugin.so:term_exit
```

Nach dem Abspeichern wird die veränderte Datei wieder in das Verzeichnis kopiert. Es ist ratsam, eine Sicherheitskopie der httpd.conf – Datei anzulegen, um bei eventuell auftretenden Fehler den Originalzustand wieder herstellen zu können.

Die Änderungen werden erst nach einem Neustart des HttpServers wirksam. Dies geschieht

mit der Hilfe des QWS3270 – Emulators. Nach einem erfolgreichen Einloggen, wird der gewohnte Menübildschirm sichtbar. Durch Eingabe des Parameters m wird das erweiterte Menu sichtbar. Das Starten und Stoppen von Jobs läßt sich von der *Spool Search and Display Facility* (SDSF) aus steuern. Durch das Kommando **5** gelangt man dort hin. In Abbildung 7.7 wird der dann erscheinende Auswahlbildschirm angezeigt.

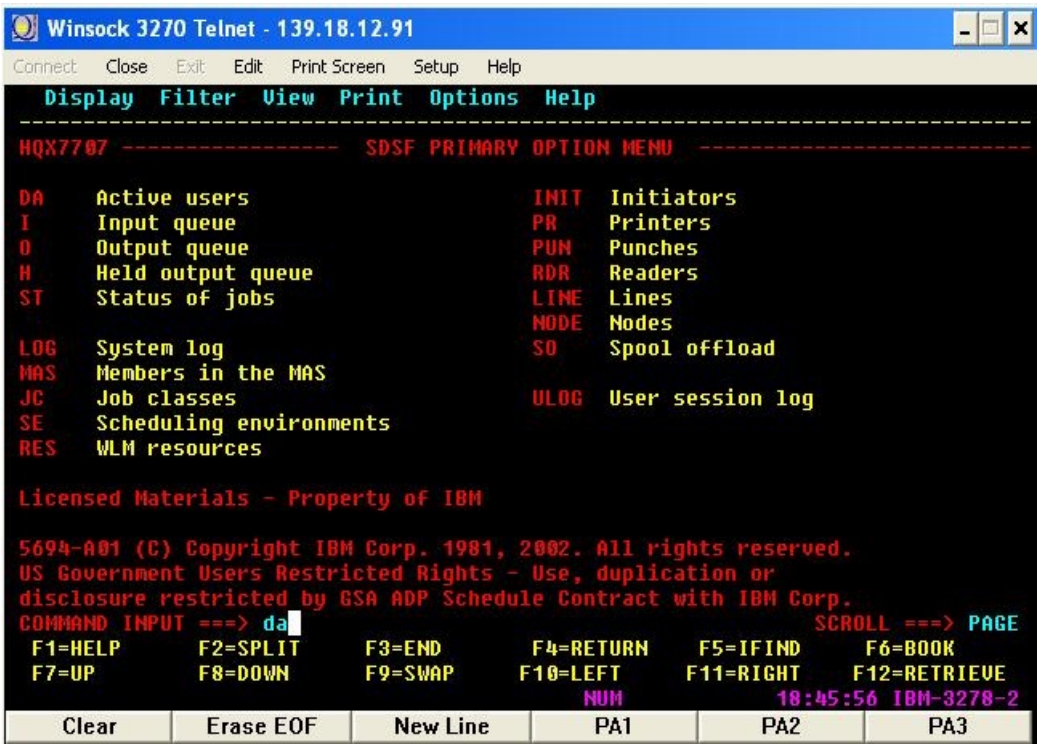


Abbildung 7.7 : SDSF Primary Option Menu

Der Punkt *Active users* stellt alle sich in Ausführung befindliche Jobs dar. Für einen Neustart des HttpServers muss der Job HTTPD1 gestoppt und anschließend wieder gestartet werden. Dazu dienen die Befehle **/P HTTPD1** (*purge* engl. = säubern) und **/S HTTPD1**. Es ist ratsam, nach der Eingabe eines Kommandos eine Weile zu warten, da die Prozeduren einige Zeit benötigen können. Es gibt zwei Möglichkeiten, um festzustellen, ob der Server erfolgreich neu gestartet wurde. Erstens kann in der Jobliste nachgeschaut werden, ob der Job HTTPD1 wieder darin aufgelistet ist und zweitens kann über den Browser die Seite **http://techspru2.informatik.uni-leipzig.de** aufgerufen werden. Wenn der Server läuft, wird die Startseite des IBM HttpServers angezeigt.

7.2.2 Einrichten einer Datenbankverbindung

Für den Fall, dass für den WebSphere Application Server noch keine Verbindung zu der DB2 – Datenbank eingerichtet wurde, wird das in diesem Abschnitt erläutert. Dazu wird der SMUI auf gewohnte Weise gestartet und eine neue *Conversation* erstellt. Unter dem Punkt **J2EE Resources** besteht die Möglichkeit, eine neue Datenbankverbindung zu definieren.

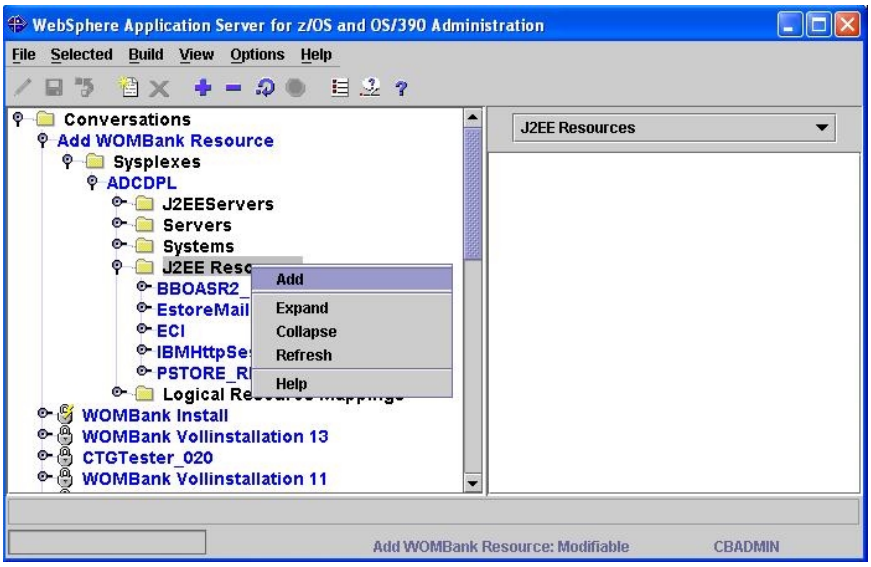


Abbildung 7.8 : Add Resource im SMUI

Eine Eingabemaske erscheint auf der rechten Seite des Fensters, in der lediglich der Name und der Typ der Quelle angegeben wird. In Abbildung 7.9 sind die erforderlichen Einstellungen dargestellt, die aus einem einfachen Pulldown – Menü ausgewählt werden können.

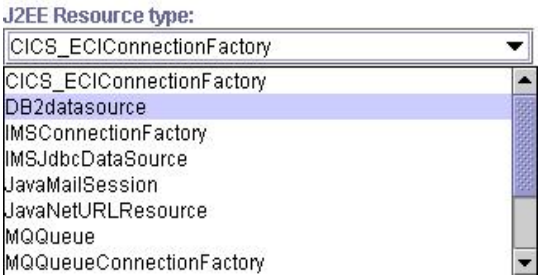


Abbildung 7.9 : Wahl des Resource Type

Nach dem Speichern der Angaben erscheint ein neuer Punkt unter J2EE Resources mit dem eben eingegebenen Namen. Darunter verbirgt sich der Unterpunkt **J2EE Resource Instances.** Damit kann für ein bestimmtes System eine Datenbankverbindung eingerichtet

werden.

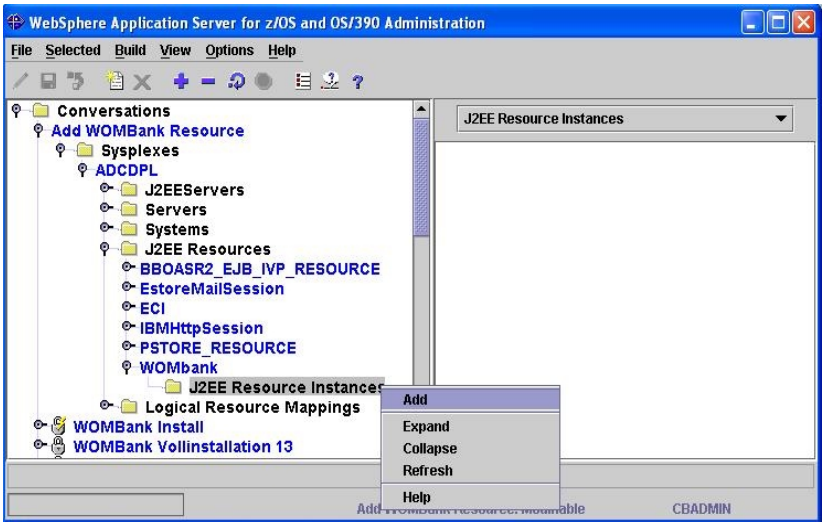


Abbildung 7.10 : Add J2EE Resource Instance

Nachdem die folgenden Parameter eingestellt wurden, steht die Verbindung zur Verfügung.

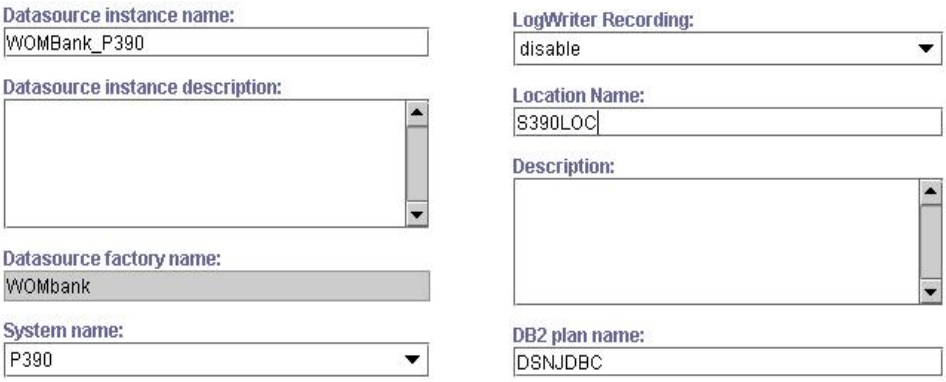


Abbildung 7.11 : Einstellungen für Resource Instance

Selbstverständlich muss die *Conversation* noch bestätigt (*commit*) und aktiviert werden.

7.2.3 Ausführen des PrepareServlets

Ist die WOMBank – Anwendung auf dem WAS installiert und die Tabellen in der Datenbank erstellt, bleibt nur noch eines zu tun. Die für die Anwendung wichtigen HTML – Dokumente müssen in die Datenbank geladen und die WBNumber – Entity Bean vorbereitet werden. Diese Aufgaben übernimmt das PrepareServlet, welches dem Installationsumfang der

WOMBank beiliegt. Bevor allerdings das Servlet seinen Dienst verrichten kann, müssen die Dateien **info.html**, **register.html**, **signon.html, new.html, exist.html, tlog.html, index.html** und **enter.html** via FTP auf den z/OS – Rechner kopiert und danach mit dem *chmod* – Befehl (*chmod 777 *.**) für alle zugänglich gemacht werden. Das betrifft auch das Verzeichnis, in dem sich die Dateien befinden.

Nun kann das Servlet mit der folgenden URL aufgerufen werden :

http://techspru2.informatik.uni-leipzig.de/WOMBank/PrepareServlet

Danach erscheint folgende HTML – Seite im Browserfenster.

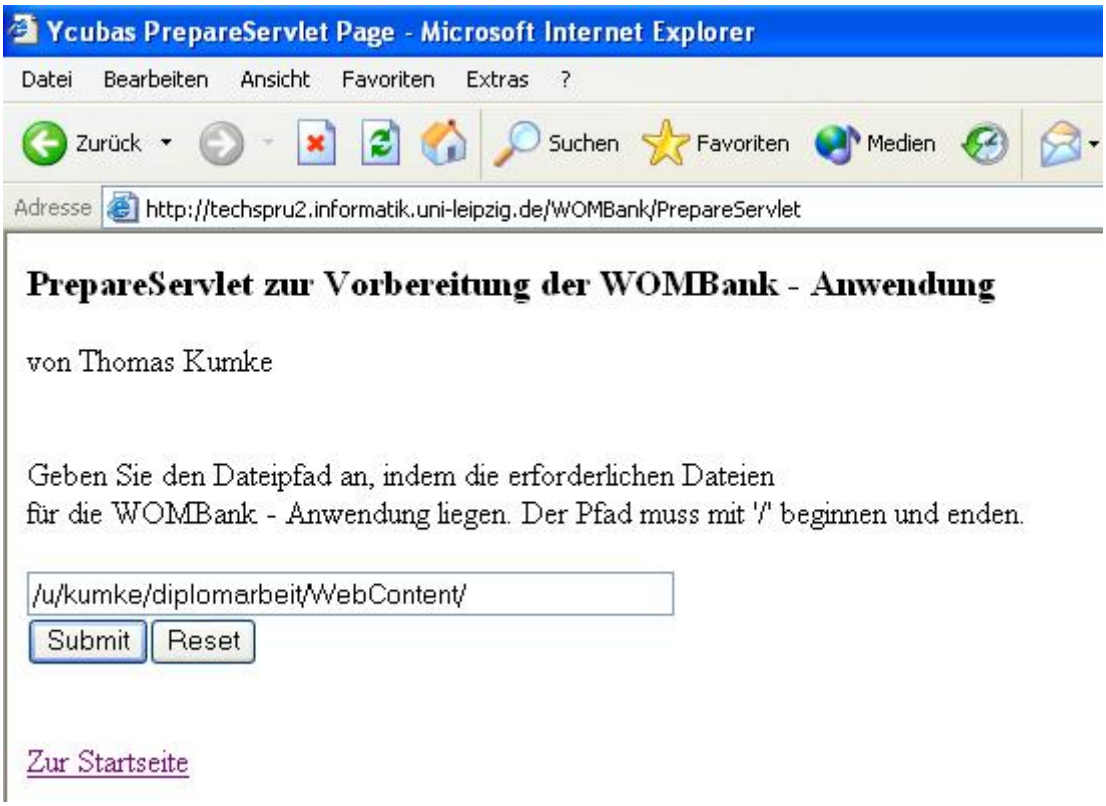


Abbildung 7.12 : PrepareServlet mit Pfadangabe

Mittels einer Eingabemaske wird das Verzeichnis angegeben, in dem sich die HTML – Dokumente befinden. Das PrepareServlet sucht in dem Verzeichnis nach den oben erwähnten Dateien und überträgt sie mit Hilfe der ByteContainer – Entity Bean in die Datenbank. Des Weiteren wird nach den zwei Datensätzen mit den Primärschlüsseln

bankaccount und *alogrecord* in der Tabelle WBNumber gesucht und sollten diese nicht gefunden werden, übernimmt das Servlet die Erstellung. Wenn alles erfolgreich abgeschlossen ist, wird eine Infoseite im Browserfenster angezeigt.

7.3 Test der WOMBank

Nach der erfolgreichen Installation der WOMBank – Anwendung sollte nun noch getestet werden, ob alles reibungslos funktioniert. Dazu noch eine kurze Erläuterung, wie sich die aufzurufende URL zusammensetzt. Eine Beschreibung dieser Zusammenhänge wurde auch schon in Kapitel 5 geliefert.

In der *application.xml* der WOMBank.ear – Datei wird der Name der Anwendung mit der Zeile

```
<module>
   <web>
     <web-uri>WOMBankOS390WAR.war</web-uri>
     <context-root>/WOMBank</context-root>
   </web>
</module>
```

festgelegt. Das ist auch genau der Name, der in der httpd.conf – Datei eingetragen werden muss, um dem HttpServer mitzuteilen, dass diese Anfrage an den WebSphere Application Server weiterzuleiten ist. Alle weiteren Pfadangaben sind in der *web.xml* zu finden. Das WOMBankServlet und alle anderen TestServlets bekommen dort einen eindeutigen Namen zugeordnet. Das sieht zum Beispiel so aus:

```
<servlet-mapping>
   <servlet-name>WOMBankServlet</servlet-name>
   <url-pattern>/WOMBankServlet</url-pattern>
</servlet-mapping>
```

7.3.1 TestServlets

Es gibt verschiedene TestServlets, die währen der Entwicklung der Anwendung entstanden sind, um den Fortschritt zu begutachten und um eventuell auftretende Fehler besser ausmachen zu können. Die Namen dieser kleinen Java – Anwendungen sind WBEntityTest, WBSessionTest und WBTest.

Das WBEntityTest – Servlet beschäftigt sich mit den EntityBeans. Dabei werden alle Tabellen in der Datenbank mit Hilfe der Beans angesprochen, Daten gespeichert und wieder ausgelesen. Die erzeugten Datensätze werden anschliessend wieder gelöscht. Über folgende URL wird der Test eingeleitet:

http://techspru2.informatik.uni-leipzig.de/WOMBank/WBEntityTest

Das WBSessionTest – Servlet überprüft die Methoden der Session Beans. Dabei werden verschiedene Funktionen der WOMBank getestet, wie zum Beispiel Konto erstellen, Geld einzahlen, Geld abheben und Geld transferieren. Der Test der Session Beans wird über folgende URL eingeleitet:

http://techspru2.informatik.uni-leipzig.de/WOMBank/WBSessionTest

Das WBTest – Servlet ist eine menügeführte Variante der TestServlets. Dabei werden verschiedene Funktionalitäten der WOMBank – Anwendung getestet. Mit der folgenden URL wird der Startbildschirm des Servlets aufgerufen:

http://techspru2.informatik.uni-leipzig.de/WOMBank/WBTest

Durch Eingabe einer Zahl wird die entsprechende Prozedur ausgeführt.

7.3.2 Ausführen der Anwendung

Nun steht der Benutzung der WOMBank nichts mehr im Wege. Die folgende URL ruft die Startseite der Anwendung auf, die auch in Abbildung 7.13 dargestellt wird:

http://techspru2.informatik.uni-leipzig.de/WOMBank/index.html

Mit dieser Seite hat man Zugriff auf alle wichtigen Funktionen der WOMBank.
Im unteren Teil des Bildschirms befinden sich die Buttons *explanation* und *code*. Damit lassen sich eine kurze Erläuterung der Menüpunkte und der Quellcode der wichtigsten Teile der Anwendung abrufen. Dazwischen befindet sich ein Link mit dem Namen *JavaDoc.* Dadurch wird eine Onlinedokumentation im JavaDoc – Stil aufgerufen, die die Enterprise Beans und deren Methoden auflistet und kurz beschreibt.

Im oberen Teil der Startseite befindet sich eine Menüleiste, die ständig angezeigt wird. Damit lassen sich alle Funktionen der Applikationen steuern und aufrufen.
Im mittleren Teil werden alle Informationen angezeigt. Das sind zum Beispiel die kundenspezifischen Kontodaten, das Logbuch der ausgeführten Aktionen oder allgemeine Informationen, die zur Erläuterung der WOMBank – Funktionalität dienen.

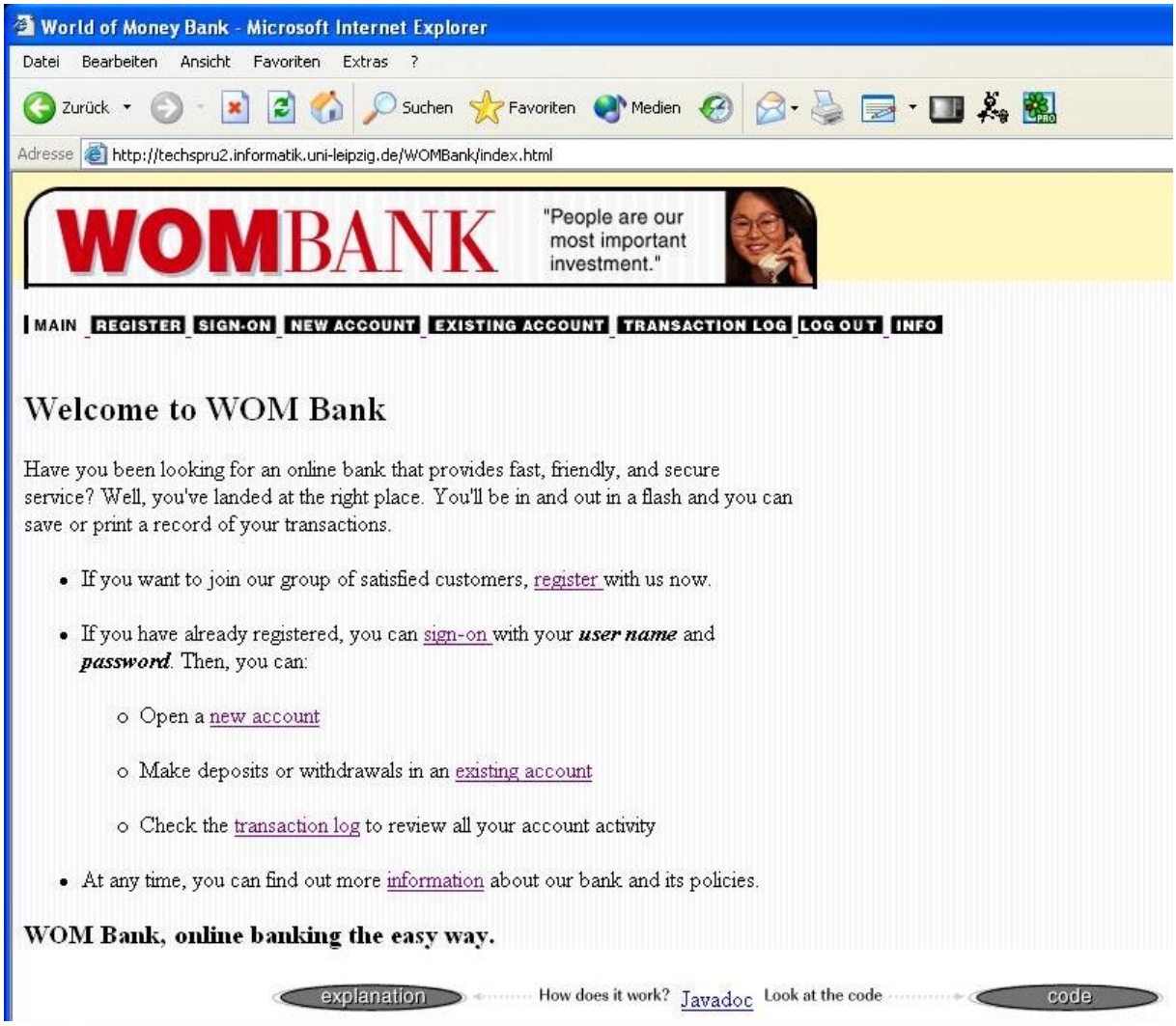


Abbildung 7.13 : Startseite der WOMBank

8 Für die Zukunft ...

Mit den bisherigen Beschreibungen sollte es möglich sein, eine EJB – Anwendung, besonders die WOMBank und die HelloWorld – Applikation, auf dem WebSphere Application Server zu installieren und zum Laufen zu bringen. Alle Konfigurationsschritte und genaue Beschreibungen sind außerdem in den zwei Tutorien zu finden, welche neben dieser Diplomarbeit entstanden sind.
Im Folgenden werden einige Anregungen geliefert, um das Thema Enterprise Java Beans in Verbindung mit dem z/OS – Betriebssystem und dem WAS in weitere Projekte einfliessen zu lassen.

8.1 Überarbeitung der WOMBank – Anwendung

Die WOMBank ist die Umsetzung eines sehr einfachen Onlinebankenkonzeptes. Um die Faktoren Sicherheit, Datenschutz und Transaktionsatomizität zu gewährleisten, müßten viele Veränderungen an der bestehenden Implementierung vorgenommen genommen werden. Ein Projekt könnte die Möglichkeiten untersuchen, wie diese Erweiterungen in die WOMBank – Applikation integriert werden können oder ob das überhaupt möglich ist. Die Programmiersprache Java bietet dafür interessante Ansätze zu den Themen Verschlüsselung und Loginmanagement.

8.2 Umsetzung einer anderen Anwendung mit Hilfe von EJBs

Im Rahmen der Diplomarbeit „Internet – basierte Anwendungen mit Java und DB2 unter OS/390“ von Herrn Ralf Ronneburger wurden mehrere Java – Anwendungen auf dem OS/390 – Rechner der Universität Leipzig installiert. Unter anderem auch die ursprüngliche

Version der WOMBank. Ein neues Projekt könnte sich damit befassen, eine andere Java – Anwendung aus der Diplomarbeit auf J2EE – Architektur umzustellen. Zwei Applikationen würden sich dafür besonders qualifizieren, TicketCentral und XtremeTravel.
Ein größeres Projekt wäre die Konzeption und Implementierung einer komplett neu entwickelten Anwendung, um zum Beispiel die Leistungsfähigkeit von Enterprise Java Beans zu testen oder um festzustellen, ob bei der Verwendung von EJBs im Vergleich zu Servlets ein Performanceverlust auftritt.

8.3 Verwendung von EJBs für den Übungsbetrieb

Im Rahmen dieser Diplomarbeit sind zwei Tutorien entstanden, die den Umgang mit EJBs in Verbindung mit dem IBM WebSphere Application Server erläutern. Um die Übungen in vollem Umfang durchführen zu können, ist es an manchen Stellen notwendig, dass der Nutzer über Administratorrechte verfügt. Es wäre von Vorteil, eine Übungsumgebung zu schaffen, in der jeder User die nötigen Rechte besitzt, um die beiden Tutorien erfolgreich abzuarbeiten.

8.4 Portierung einer EJB - Anwendung auf J2EE Version 1.4

Der Universität steht bald ein neuer IBM Rechner der *zSeries* zur Verfügung, auf dem das Betriebssystem z/OS in der Version 1.5 installiert ist. Damit steht der IBM WebSphere Application Server 5 zur Verfügung, der die EJB – 2.0 – Spezifikation voll unterstützt. Die Verbesserungen dieser Version könnten in bestehende Webanwendungen integriert werden . In Abschnitt 3.3.9 werden die Neuerungen kurz erläutert.
So könnten zum Beispiel lokale Interfaces verwendet werden, um den Kommunikationsaufwand zu minimieren, der bei der Verwendung von Client – Interfaces entsteht. Weiterhin könnten Abhängigkeiten von Tabellen in der Datenbank mit dem Konzept der **Container Managed Relations** umgesetzt werden.

9 Quellenverzeichnis

[1] William Crawford, JimFarley, David Flanagan : *Java Enterprise in a Nutshell*. Deutsche Übersetzung Jörg Staudemeyer & Sascha Kersken, O'REILLY, 1.Auflage Dezember 2002, ISBN 3-89721-334-6

[2] Paul Herrmann, Udo Kebschull, Wilhelm G. Spruth: *Einführung in z/OS und OS/390.* Oldenbourg Verlag, 2003, ISBN 3-486-27214-4

[3] Volker Turau, Ronald Pfeiffer : *Java Server Pages Dynamische Generierung von Web – Dokumenten.* dpunkt.verlag, 2001, ISBN 3-89864-131-7

[4] Roland Trauner, Denis Gaebler, Bruce Smith : *OS/390 e-business Infrastructure: IBM Websphere Application Server 1.2 Customization and Usage.* IBM Redbook SG24-5604-01, 2000, ISBN 0-73841-901-X,
URL : http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245604.pdf

[5] Ralf Ronneburger : Diplomarbeit : *Internet – basierte Anwendungen mit Java und DB2 unter OS/390* , April 2003

[6] Andreas Zientek : Diplomarbeit : *z/OS & WebSphere Application Server 4.0 Installation und Konfiguration* . September 2003

[7] IBM : *WebSphere Application Server V4.0.1 for z/OS and OS/390 : System Management User Interface*, SA22-7838-04, Juli 2003
URL :
ftp://ftp.software.ibm.com/software/webserver/appserv/zos_os390/v401/boss1mst.pdf

[8] IBM : *WebSphere Application Server V4.0.1 for z/OS and OS/390 : Operations and Administration*, SA22-7835-05, Juli 2003
URL :
ftp://ftp.software.ibm.com/software/webserver/appserv/zos_os390/v401/bosb1mst.pdf

[9] IBM : *WebSphere Application Server V4.0.1 for z/OS and OS/390 : Assembling Java™ 2 Enterprise Edition (J2EE) Applications*, SA22-7836-06, Juli 2003
URL : ftp://ftp.software.ibm.com/software/webserver/appserv/zos_os390/v401/bosc1mst.pdf

[10] IBM : *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*. IBM Redbook SG24-6283-00
URL : http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246283.pdf

[11] Robert Berger, Martin Pirklbauer, Martin Wollendorfer : *Servlets, JSP, Java Beans and EJB's* April 2002
URL : http://www.iicm.edu/cguett/education/projects/javatech2002/seminar.pdf

[12] Jan Steinkraus : *EJB – Sicherheit Nach J2EE – Spezifikation 1.4*, 2003
URL : http://www.inf.fu-berlin.de/lehre/WS02/sss/slides/EJB-Ausarbeitung.pdf

[13] Clemens Dorda : *Application – Server – Techniken für Web – Anwendungen.* URL : http://www.informatik.uni-stuttgart.de/ipvv/as/lehre/hauptseminar/docws99/paper06.pdf

[14] Clemens Dorda : *Verwaltung von Benutzerprofilen mit Enterprise Java Beans*
URL : http://elib.uni-stuttgart.de/opus/volltexte/2001/768/pdf/Stud-1799.pdf

[15] Sebastian Lempert, Hauke Traulsen : *Enterprise JavaBeans™* 2.0. Juli 2002, URL : http://www.inf.fu-berlin.de/lehre/SS02/KBSE/doc/ejb2/Enterprise%20JavaBeans%202.0.pdf

[16] Marion Beckers : *Java Servlets Eine Einführung* . 2002,
URL : http://www-i3.informatik.rwth-aachen.de/teaching/02/proseminar/prosem-servlets.pdf

[17] Peter Köller : *Servlets und JavaServer-Pages* . 2000,
URL : http://www.metaprojekt.de/Downloads/servlets.pdf

[18] Sönke Paul : *Applicationserver* , URL : http://www.db.informatik.uni-kassel.de/lehre/WS0304/SeminarPI/saul/Applicationserver.pdf

[19] Dirk Lucht : *Erfahrungen bei der Durchführung eines J2EE – Projektes Verteilte Anwendungen in der Praxis (1)* .
URL : http://www.cs-consulting.net/downloads/artikel/10-J2EE-Teil.pdf

[20] Client / Server – Systeme, URL : http://wwwdvs.informatik.uni-kl.de/courses/TAS/SS2003/Vorlesungsunterlagen/Kapitel.02.half.pdf

[21] Netplanet, Homepage. URL : http://www.netplanet.org

[22] Sun Java, Homepage . URL : http://java.sun.com

[23] IBM WebSphere, Homepage . URL : http://www.ibm.com/websphere

[24] Vienna University of Technology, Homepage . URL : http://gd.tuwien.ac.at/study

10 Abbildungsverzeichnis

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Merseburg , 28. Februar 2005

A CD - Verzeichnis

Die gesamte Diplomarbeit im PDF – und OpenOffice – Format, alle Grafiken, erwähnte

Software, die zwei erarbeiteten Tutorien und die Literaturreferenzen befinden sich auf der CD, die dieser Arbeit beiliegt. Eine **index.html** – Datei bietet einen Einstiegspunkt und kann vom Hauptverzeichnis auf der CD abgerufen werden. Alle weiteren Daten befinden sich im Verzeichnis **\Diplom Thomas Kumke** .

Die Verzeichnisstruktur der CD:

\Diplom Thomas Kumke

\Diplomarbeit

\Arbeit	enthält die Niederschrift der Arbeit in PDF und OpenOffice - Format
\Images	enthält alle Grafiken im JPEG - oder OpenOffice - Format, die in der Arbeit verwendet werden
\Literatur	enthält die meisten der Quellen in PDF

\Diplomapplikationen

\bld\Diplomarbeit	enthält die compilierten Class - Dateien in einer Verzeichnisstruktur
\jar	enthält wichtige Dateien für die Erstellung der EAR - Archive
\javadoc\Diplomarbeit	enthält alle JavaDoc - Files für die Onlinedokumentation
\lib\Diplomarbeit	enthält die Archive (JAR, WAR, EAR)
\src\Diplomarbeit	enthält alle Quelldateien, Deployment Deskriptoren, den WebContent (HTML - Dateien) und die Stapelverarbeitungsdatei, mit der die Archive gebaut werden können

\DB2 Tabellen enthält die Datei createTable, zum Erstellen der WOMBank - Tabellen und die Datei dropTable zum Löschen der Tabellen

\Tutorien

\Images	enthält alle Grafiken im JPEG - Format, die in den beiden Tutorien verwendet wurden
\Tutorium 19	enthält das Tutorium 19 in PDF und OpenOffice
\Tutorium 20	enthält das Tutorium 20 in PDF und OpenOffice

\Tutoriumapplikationen

\bld\Tutorium	enthält die compilierten Class - Dateien in einer Verzeichnisstruktur
\jar	enthält wichtige Dateien für die Erstellung der

EAR -	Archive
\lib\Tutorium	enthält die Archive (JAR, WAR, EAR)
\src\Tutorium	enthält alle Quelldateien, Deployment Deskriptoren, den WebContent (HTML - Dateien) und die Stapelverarbeitungsdatei, mit der die Archive gebaut werden können

\Software

\Assembly Tool	enthält die Installationsdatei des Assembly Tools
\J2SDK	enthält die Installationsdatei des Java Software Development Kits in der Version 1.4.2_07
\QWS3270	enthält die Installationsdatei des 3270 - Emulators
\OpenOffice	enthält die Installationsdatei für OpenOffice 1.3
\WAS-Tools	enthält die Installationsdatei für den SMUI - Client
\WS_FTP	enthält die Installationsdatei des FTP - Programms